

Arm®v8-M Architecture Reference Manual

The logo for Arm, consisting of the lowercase letters 'arm' in a bold, sans-serif font.

Release information

Date	Version	Changes
02/10/2019	A.m Non-confidential-EAC	<ul style="list-style-type: none">• Eleventh EAC release
28/06/2019	A.l Non-confidential-EAC	<ul style="list-style-type: none">• Tenth EAC release
29/03/2019	A.k Non-confidential-EAC	<ul style="list-style-type: none">• Ninth EAC release
14/12/2018	A.j Non-confidential-EAC	<ul style="list-style-type: none">• Eighth EAC release
29/06/2018	A.i Non-confidential-EAC	<ul style="list-style-type: none">• Seventh EAC release
31/03/2018	A.h Non-confidential-EAC	<ul style="list-style-type: none">• Sixth EAC release
15/12/2017	A.g Non-confidential-EAC	<ul style="list-style-type: none">• Fifth EAC release
29/09/2017	A.f Non-confidential-EAC	<ul style="list-style-type: none">• Fourth EAC release
02/06/2017	A.e Non-confidential-EAC	<ul style="list-style-type: none">• Third EAC release
30/11/2016	A.d Non-confidential-EAC	<ul style="list-style-type: none">• Second EAC release
30/09/2016	A.c Non-confidential-EAC	<ul style="list-style-type: none">• EAC release
28/07/2016	A.b Non-confidential-Beta	<ul style="list-style-type: none">• Beta release
29/03/2016	A.a Confidential-Beta	<ul style="list-style-type: none">• Beta release, limited circulation

Armv8-M Architecture Reference Manual

Copyright © 2015 - 2019 Arm Limited or its affiliates. All rights reserved. The copyright statement reflects the fact that some draft issues of this document have been released, to a limited circulation.

Proprietary Notice

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

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

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

This document may include technical inaccuracies or typographical errors.

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

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

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

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

Copyright © 2015 - 2019 Arm Limited (or its affiliates). All rights reserved.

Arm Limited. Company 02557590 registered in England.

110 Fulbourn Road, Cambridge, England CB1 9NJ.

LES-PRE-20349

Confidentiality Status

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

Product Status

The information in this document is final, that is for a developed product.

Web Address

<http://www.arm.com>

Contents

Arm® v8-M Architecture Reference Manual

Release information	ii
Armv8-M Architecture Reference Manual	iii
Proprietary Notice	iii
Confidentiality Status	iii
Product Status	iv
Web Address	iv

Preface

About this book	xxix
Using this book	xxx
Conventions	xxxii
Typographical conventions	xxxii
Signals	xxxii
Numbers	xxxiii
Pseudocode descriptions	xxxiii
Assembler syntax descriptions	xxxiii
Additional reading	xxxiv
Arm publications	xxxiv
Other publications	xxxiv
Feedback	xxxv
Feedback on this book	xxxv

Part A Armv8-M Architecture Introduction and Overview

Chapter A1

Introduction

A1.1 Document layout and terminology	38
A1.1.1 Structure of the document	38
A1.1.2 Scope of the document	39
A1.1.3 Intended audience	39
A1.1.4 Terminology, phrases	39
A1.1.5 Terminology, Armv8-M specific terms	40
A1.2 About the Armv8 architecture, and architecture profiles	41
A1.3 The Armv8-M architecture profile	42
A1.3.1 Security Extension	42
A1.3.2 MPU model	42
A1.3.3 Nested Vector Interrupt Controller	42
A1.3.4 Stack pointers	42
A1.3.5 The Armv8-M instruction set	43
A1.3.6 Debug	43
A1.4 Armv8-M variants	44

Part B Armv8-M Architecture Rules

Chapter B1

Resets

B1.1 Resets, Cold reset, and Warm reset	47
---	----

Chapter B2

Power Management

B2.1	Power management	49
B2.1.1	The Wait for Event (WFE) instruction	49
B2.1.2	The Event register	49
B2.1.3	The Wait for Interrupt (WFI) instruction	50
B2.2	Sleep on exit	51

Chapter B3

Programmers' Model

B3.1	PE modes, Thread mode and Handler mode	54
B3.2	Privileged and unprivileged execution	55
B3.3	Registers	56
B3.4	Special-purpose CONTROL register	58
B3.5	XPSR, APSR, IPSR, and EPSR	59
B3.5.1	Interrupt Program Status Register (IPSR)	59
B3.5.2	Execution Program Status Register (EPSR)	60
B3.6	Security states: Secure state, and Non-secure state	61
B3.7	Security states and register banking between Security states	62
B3.8	Stack pointer	63
B3.9	Exception numbers and exception priority numbers	65
B3.10	Exception enable, pending, and active bits	68
B3.11	Security states, exception banking	70
B3.12	Faults	72
B3.13	Priority model	77
B3.14	Secure address protection	81
B3.15	Security state transitions	82
B3.16	Function calls from Secure state to Non-secure state	84
B3.17	Function returns from Non-secure state	85
B3.18	Exception handling	87
B3.19	Exception entry, context stacking	89
B3.20	Exception entry, register clearing after context stacking	97
B3.21	Stack limit checks	98
B3.22	Exception return	101
B3.23	Integrity signature	104
B3.24	Exceptions during exception entry	105
B3.25	Exceptions during exception return	107
B3.26	Tail-chaining	108
B3.27	Exceptions, instruction resume, or instruction restart	111
B3.28	Vector tables	114
B3.29	Hardware-controlled priority escalation to HardFault	116
B3.30	Special-purpose mask registers, PRIMASK, BASEPRI, FAULTMASK, for configurable priority boosting	117
B3.31	Lockup	119
B3.31.1	Instruction-related lockup behavior	119
B3.31.2	Exception-related lockup behavior	121
Errors when unstacking state on exception return	123	
B3.32	Context Synchronization Event	124
B3.33	Coprocessor support	125

Chapter B4

Floating-point Support

B4.1	The optional Floating-point Extension, FPv5	127
B4.2	About the Floating-point Status and Control Registers	129
B4.3	Registers for Floating-point data processing, S0-S31, or D0-D15	130
B4.4	Floating-point standards and terminology	131
B4.5	Floating-point data representable	132
B4.6	Floating-point encoding formats, half-precision, single-precision, and double-precision	133

B4.7	The IEEE 754 Floating-point exceptions	135
B4.8	The Flush-to-zero mode	136
B4.9	The Default NaN mode, and NaN handling	138
B4.10	The Default NaN	139
B4.11	Combinations of Floating-point exceptions	140
B4.12	Priority of Floating-point exceptions relative to other Floating-point exceptions	141

Chapter B5

Memory Model

B5.1	Memory accesses	144
B5.2	Address space	145
B5.3	Endianness	146
B5.4	Alignment behavior	148
B5.5	Atomicity	149
B5.5.1	Single-copy atomicity	149
B5.5.2	Multi-copy atomicity	149
B5.6	Concurrent modification and execution of instructions	151
B5.7	Access rights	153
B5.8	Observability of memory accesses	155
B5.9	Completion of memory accesses	157
B5.10	Ordering requirements for memory accesses	158
B5.11	Ordering of implicit memory accesses	159
B5.12	Ordering of explicit memory accesses	160
B5.13	Memory barriers	161
B5.13.1	Instruction Synchronization Barrier	161
B5.13.2	Data Memory Barrier	161
B5.13.3	Data Synchronization Barrier	162
B5.13.4	Consumption of Speculative Data Barrier	163
B5.13.5	Physical Speculative Store Bypass Barrier	164
B5.13.6	Speculative Store Bypass Barrier	164
B5.13.7	Synchronization requirements for System Control Space	164
B5.14	Normal memory	166
B5.15	Cacheability attributes	168
B5.16	Device memory	169
B5.17	Device memory attributes	171
B5.17.1	Gathering and non-Gathering Device memory attributes	172
B5.17.2	Reordering and non-Reordering Device memory attributes	172
B5.17.3	Early Write Acknowledgement and no Early Write Acknowledgement Device memory attributes	173
B5.18	Shareability domains	174
B5.19	Shareability attributes	176
B5.20	Memory access restrictions	177
B5.21	Mismatched memory attributes	178
B5.22	Load-Exclusive and Store-Exclusive accesses to Normal memory	180
B5.23	Load-Acquire and Store-Release accesses to memory	181
B5.24	Caches	183
B5.25	Cache identification	185
B5.26	Cache visibility	186
B5.27	Cache coherency	187
B5.28	Cache enabling and disabling	188
B5.29	Cache behavior at reset	189
B5.30	Behavior of Preload Data (PLD) and Preload Instruction (PLI) instructions with caches	190
B5.31	Branch predictors	191
B5.32	Cache maintenance operations	192
B5.33	Ordering of cache maintenance operations	196

	B5.34	Branch predictor maintenance operations	197
Chapter B6		The System Address Map	
	B6.1	System address map	199
	B6.2	The System region of the system address map	200
	B6.3	The System Control Space (SCS)	202
Chapter B7		Synchronization and Semaphores	
	B7.1	Exclusive access instructions	204
	B7.2	The local monitors	205
	B7.3	The global monitor	207
		B7.3.1 Load-Exclusive and Store-Exclusive	208
		B7.3.2 Load-Exclusive and Store-Exclusive in Shareable memory	209
	B7.4	Exclusive access instructions and the monitors	211
	B7.5	Load-Exclusive and Store-Exclusive instruction constraints	212
Chapter B8		The Armv8-M Protected Memory System Architecture	
	B8.1	Memory Protection Unit	215
	B8.2	Security attribution	218
	B8.3	Security attribution unit (SAU)	221
	B8.4	IMPLEMENTATION DEFINED Attribution Unit (IDAU)	222
Chapter B9		The System Timer, SysTick	
	B9.1	The system timer, SysTick	224
Chapter B10		Nested Vectored Interrupt Controller	
	B10.1	NVIC definition	227
	B10.2	NVIC operation	228
Chapter B11		Debug	
	B11.1	Debug feature overview	231
		B11.1.1 Debug mechanisms	233
		B11.1.2 Debug resources	233
		B11.1.3 Trace	234
	B11.2	Accessing debug features	236
		B11.2.1 ROM table	236
		B11.2.2 Debug System registers	238
		B11.2.3 CoreSight and identification registers	238
	B11.3	Debug authentication interface	240
		B11.3.1 Halting debug authentication	241
		B11.3.2 Non-invasive debug authentication	242
		B11.3.3 DebugMonitor exception authentication	243
		B11.3.4 DAP access permissions	245
	B11.4	Debug event behavior	250
		B11.4.1 About debug events	250
		B11.4.2 Debug stepping	255
		B11.4.3 Vector catch	259
		B11.4.4 Breakpoint instructions	261
		B11.4.5 External debug request	261
	B11.5	Debug state	263
	B11.6	Exiting Debug state	266
	B11.7	Multiprocessor support	267
		B11.7.1 Cross-halt event	267
		B11.7.2 External restart request	267
Chapter B12		Debug and Trace Components	

B12.1	Instrumentation Trace Macrocell	269
B12.1.1	About the ITM	269
B12.1.2	ITM operation	270
B12.1.3	Timestamp support	272
B12.1.4	Synchronization support	276
B12.1.5	Continuation bits	276
B12.2	Data Watchpoint and Trace unit	278
B12.2.1	About the DWT	278
B12.2.2	DWT unit operation	279
B12.2.3	Constraints on programming DWT comparators	282
B12.2.4	CMPMATCH trigger events	286
B12.2.5	Matching in detail	286
B12.2.6	DWT match restrictions and relaxations	289
B12.2.7	DWT trace restrictions and relaxations	291
B12.2.8	CYCCNT cycle counter and related timers	292
B12.2.9	Profiling counter support	294
B12.2.10	Program Counter sampling support	296
B12.3	Embedded Trace Macrocell	298
B12.4	Trace Port Interface Unit	299
B12.5	Flash Patch and Breakpoint unit	301
B12.5.1	About the FPB unit	301
B12.5.2	FPB unit operation	301
B12.5.3	Cache maintenance	303

Part C Armv8-M Instruction Set

Chapter C1

Instruction Set Overview

C1.1	Instruction set	306
C1.2	Format of instruction descriptions	307
C1.2.1	The title	307
C1.2.2	A short description	307
C1.2.3	The instruction encoding or encodings	307
C1.2.4	Any alias conditions, if applicable	308
C1.2.5	Standard assembler syntax fields	309
C1.2.6	Pseudocode describing how the instruction operates	310
C1.2.7	Use of labels in UAL instruction syntax	311
C1.2.8	Using syntax information	312
C1.3	Conditional execution	313
C1.3.1	Conditional instructions	314
C1.3.2	Pseudocode details of conditional execution	314
C1.3.3	Conditional execution of undefined instructions	314
C1.3.4	Interaction of undefined instruction behavior with UNPREDICTABLE or CONSTRAINED UNPREDICTABLE instruction behavior	315
C1.3.5	ITSTATE	315
C1.3.6	Pseudocode details of ITSTATE operation	316
C1.3.7	SVC and ISTATE	316
C1.3.8	CONSTRAINED UNPREDICTABLE behavior and IT blocks	316
C1.4	Instruction set encoding information	319
C1.4.1	UNDEFINED and UNPREDICTABLE instruction set space	319
C1.4.2	Pseudocode descriptions of operations on general-purpose registers and the PC	319
C1.4.3	Use of 0b1111 as a register specifier	319
C1.4.4	Use of 0b1101 as a register specifier	321
C1.4.5	16-bit T32 instruction support for SP	322

C1.4.6	Branching	322
C1.4.7	Instruction set, interworking and interstating support	323
C1.5	Modified immediate constants	324
C1.5.1	Operation of modified immediate constants	324
C1.6	NOP-compatible hint instructions	325
C1.7	SBZ or SBO fields in instructions	326

Chapter C2

Instruction Specification

C2.1	Top level T32 instruction set encoding	328
C2.2	16-bit T32 instruction encoding	329
C2.2.1	Shift (immediate), add, subtract, move, and compare	329
C2.2.2	Data-processing (two low registers)	331
C2.2.3	Special data instructions and branch and exchange	331
C2.2.4	Load/store (register offset)	332
C2.2.5	Load/store word/byte (immediate offset)	333
C2.2.6	Load/store halfword (immediate offset)	333
C2.2.7	Load/store (SP-relative)	333
C2.2.8	Add PC/SP (immediate)	334
C2.2.9	Miscellaneous 16-bit instructions	334
C2.2.10	Load/store multiple	336
C2.2.11	Conditional branch, and Supervisor Call	336
C2.3	32-bit T32 instruction encoding	338
C2.3.1	Load/store (multiple, dual, exclusive, acquire-release), table branch	338
C2.3.2	Data-processing (shifted register)	343
C2.3.3	Data-processing (modified immediate)	344
C2.3.4	Data-processing (plain binary immediate)	345
C2.3.5	Branches and miscellaneous control	346
C2.3.6	Load/store single	348
C2.3.7	Data-processing (register)	354
C2.3.8	Multiply, multiply accumulate, and absolute difference	357
C2.3.9	Long multiply and divide	359
C2.3.10	Coprocessor and floating-point instructions	359
C2.4	Alphabetical list of instructions	367
C2.4.1	ADC (immediate)	368
C2.4.2	ADC (register)	369
C2.4.3	ADD (SP plus immediate)	371
C2.4.4	ADD (SP plus register)	373
C2.4.5	ADD (immediate)	375
C2.4.6	ADD (immediate, to PC)	378
C2.4.7	ADD (register)	380
C2.4.8	ADR	383
C2.4.9	AND (immediate)	385
C2.4.10	AND (register)	386
C2.4.11	ASR (immediate)	388
C2.4.12	ASR (register)	390
C2.4.13	ASRS (immediate)	392
C2.4.14	ASRS (register)	394
C2.4.15	B	396
C2.4.16	BFC	398
C2.4.17	BFI	399
C2.4.18	BIC (immediate)	400
C2.4.19	BIC (register)	401
C2.4.20	BKPT	403
C2.4.21	BL	404
C2.4.22	BLX, BLXNS	405

Contents

C2.4.23	BX, BXNS	407
C2.4.24	CBNZ, CBZ	408
C2.4.25	CDP, CDP2	409
C2.4.26	CLREX	411
C2.4.27	CLZ	412
C2.4.28	CMN (immediate)	413
C2.4.29	CMN (register)	414
C2.4.30	CMP (immediate)	416
C2.4.31	CMP (register)	417
C2.4.32	CPS	419
C2.4.33	CSDB	421
C2.4.34	DBG	422
C2.4.35	DMB	423
C2.4.36	DSB	424
C2.4.37	EOR (immediate)	425
C2.4.38	EOR (register)	426
C2.4.39	FLDMDBX, FLDMIAX	428
C2.4.40	FSTMDBX, FSTMIAX	430
C2.4.41	ISB	432
C2.4.42	IT	433
C2.4.43	LDA	435
C2.4.44	LDAB	436
C2.4.45	LDAEX	437
C2.4.46	LDAEXB	438
C2.4.47	LDAEXH	439
C2.4.48	LDAH	440
C2.4.49	LDC, LDC2 (immediate)	441
C2.4.50	LDC, LDC2 (literal)	444
C2.4.51	LDM, LDMIA, LDMFD	446
C2.4.52	LDMDB, LDMEA	450
C2.4.53	LDR (immediate)	453
C2.4.54	LDR (literal)	457
C2.4.55	LDR (register)	459
C2.4.56	LDRB (immediate)	461
C2.4.57	LDRB (literal)	464
C2.4.58	LDRB (register)	465
C2.4.59	LDRBT	467
C2.4.60	LDRD (immediate)	468
C2.4.61	LDRD (literal)	470
C2.4.62	LDREX	472
C2.4.63	LDREXB	473
C2.4.64	LDREXH	474
C2.4.65	LDRH (immediate)	475
C2.4.66	LDRH (literal)	478
C2.4.67	LDRH (register)	479
C2.4.68	LDRHT	481
C2.4.69	LDRSB (immediate)	482
C2.4.70	LDRSB (literal)	484
C2.4.71	LDRSB (register)	485
C2.4.72	LDRSBT	487
C2.4.73	LDRSH (immediate)	488
C2.4.74	LDRSH (literal)	490
C2.4.75	LDRSH (register)	491
C2.4.76	LDRSHT	493
C2.4.77	LDRT	494

Contents

C2.4.78	LSL (immediate)	495
C2.4.79	LSL (register)	497
C2.4.80	LSLS (immediate)	499
C2.4.81	LSLS (register)	501
C2.4.82	LSR (immediate)	503
C2.4.83	LSR (register)	505
C2.4.84	LSRS (immediate)	507
C2.4.85	LSRS (register)	509
C2.4.86	MCR, MCR2	511
C2.4.87	MCRR, MCRR2	513
C2.4.88	MLA	515
C2.4.89	MLS	516
C2.4.90	MOV (immediate)	517
C2.4.91	MOV (register)	519
C2.4.92	MOV, MOVS (register-shifted register)	523
C2.4.93	MOVT	526
C2.4.94	MRC, MRC2	527
C2.4.95	MRRC, MRRC2	529
C2.4.96	MRS	531
C2.4.97	MSR (register)	535
C2.4.98	MUL	540
C2.4.99	MVN (immediate)	542
C2.4.100	MVN (register)	543
C2.4.101	NOP	545
C2.4.102	ORN (immediate)	546
C2.4.103	ORN (register)	547
C2.4.104	ORR (immediate)	549
C2.4.105	ORR (register)	550
C2.4.106	PKHBT, PKHTB	552
C2.4.107	PLD (literal)	554
C2.4.108	PLD, PLDW (immediate)	555
C2.4.109	PLD, PLDW (register)	557
C2.4.110	PLI (immediate, literal)	558
C2.4.111	PLI (register)	560
C2.4.112	POP (multiple registers)	561
C2.4.113	POP (single register)	563
C2.4.114	PSSBB	564
C2.4.115	PUSH (multiple registers)	565
C2.4.116	PUSH (single register)	567
C2.4.117	QADD	568
C2.4.118	QADD16	569
C2.4.119	QADD8	570
C2.4.120	QASX	571
C2.4.121	QDADD	572
C2.4.122	QDSUB	573
C2.4.123	QSAX	574
C2.4.124	QSUB	575
C2.4.125	QSUB16	576
C2.4.126	QSUB8	577
C2.4.127	RBIT	578
C2.4.128	REV	579
C2.4.129	REV16	581
C2.4.130	REVSH	583
C2.4.131	ROR (immediate)	585
C2.4.132	ROR (register)	586

C2.4.133 RORS (immediate)	588
C2.4.134 RORS (register)	589
C2.4.135 RRX	591
C2.4.136 RRXS	592
C2.4.137 RSB (immediate)	593
C2.4.138 RSB (register)	595
C2.4.139 SADD16	597
C2.4.140 SADD8	598
C2.4.141 SASX	599
C2.4.142 SBC (immediate)	600
C2.4.143 SBC (register)	601
C2.4.144 SBFX	603
C2.4.145 SDIV	604
C2.4.146 SEL	605
C2.4.147 SEV	606
C2.4.148 SG	607
C2.4.149 SHADD16	608
C2.4.150 SHADD8	609
C2.4.151 SHASX	610
C2.4.152 SHSAX	611
C2.4.153 SHSUB16	612
C2.4.154 SHSUB8	613
C2.4.155 SMLABB, SMLABT, SMLATB, SMLATT	614
C2.4.156 SMLAD, SMLADX	616
C2.4.157 SMLAL	617
C2.4.158 SMLALBB, SMLALBT, SMLALTB, SMLALTT	618
C2.4.159 SMLALD, SMLALDX	620
C2.4.160 SMLAWB, SMLAWT	622
C2.4.161 SMLSd, SMLSdX	623
C2.4.162 SMLSd, SMLSdX	624
C2.4.163 SMMLA, SMMLAR	626
C2.4.164 SMMLS, SMMLSR	627
C2.4.165 SMMUL, SMMULR	628
C2.4.166 SMUAD, SMUADX	629
C2.4.167 SMULBB, SMULBT, SMULTB, SMULTT	630
C2.4.168 SMULL	632
C2.4.169 SMULWB, SMULWT	633
C2.4.170 SMUSD, SMUSDx	634
C2.4.171 SSAT	635
C2.4.172 SSAT16	636
C2.4.173 SSAX	637
C2.4.174 SSBB	638
C2.4.175 SSUB16	639
C2.4.176 SSUB8	640
C2.4.177 STC, STC2	641
C2.4.178 STL	644
C2.4.179 STLB	645
C2.4.180 STLEX	646
C2.4.181 STLEXB	648
C2.4.182 STLEXH	650
C2.4.183 STLH	652
C2.4.184 STM, STMIA, STMEA	653
C2.4.185 STMDB, STMFD	656
C2.4.186 STR (immediate)	659
C2.4.187 STR (register)	662

Contents

C2.4.188	STRB (immediate)	664
C2.4.189	STRB (register)	667
C2.4.190	STRBT	669
C2.4.191	STRD (immediate)	670
C2.4.192	STREX	672
C2.4.193	STREXB	674
C2.4.194	STREXH	676
C2.4.195	STRH (immediate)	678
C2.4.196	STRH (register)	681
C2.4.197	STRHT	683
C2.4.198	STRT	684
C2.4.199	SUB (SP minus immediate)	685
C2.4.200	SUB (SP minus register)	687
C2.4.201	SUB (immediate)	689
C2.4.202	SUB (immediate, from PC)	692
C2.4.203	SUB (register)	693
C2.4.204	SVC	695
C2.4.205	SXTAB	696
C2.4.206	SXTAB16	697
C2.4.207	SXTAH	698
C2.4.208	SXTB	699
C2.4.209	SXTB16	701
C2.4.210	SXTH	702
C2.4.211	TBB, TBH	704
C2.4.212	TEQ (immediate)	705
C2.4.213	TEQ (register)	706
C2.4.214	TST (immediate)	707
C2.4.215	TST (register)	708
C2.4.216	TT, TTT, TTA, TTAT	710
C2.4.217	UADD16	712
C2.4.218	UADD8	713
C2.4.219	UASX	714
C2.4.220	UBFX	715
C2.4.221	UDF	716
C2.4.222	UDIV	717
C2.4.223	UHADD16	718
C2.4.224	UHADD8	719
C2.4.225	UHASX	720
C2.4.226	UHSAX	721
C2.4.227	UHSUB16	722
C2.4.228	UHSUB8	723
C2.4.229	UMAAL	724
C2.4.230	UMLAL	725
C2.4.231	UMULL	726
C2.4.232	UQADD16	727
C2.4.233	UQADD8	728
C2.4.234	UQASX	729
C2.4.235	UQSAX	730
C2.4.236	UQSUB16	731
C2.4.237	UQSUB8	732
C2.4.238	USAD8	733
C2.4.239	USADA8	734
C2.4.240	USAT	735
C2.4.241	USAT16	736
C2.4.242	USAX	737

C2.4.243	USUB16	738
C2.4.244	USUB8	739
C2.4.245	UXTAB	740
C2.4.246	UXTAB16	741
C2.4.247	UXTAH	742
C2.4.248	UXTB	743
C2.4.249	UXTB16	745
C2.4.250	UXTH	746
C2.4.251	VABS	748
C2.4.252	VADD	749
C2.4.253	VCMP	750
C2.4.254	VCMPE	752
C2.4.255	VCVT (between double-precision and single-precision)	754
C2.4.256	VCVT (between floating-point and fixed-point)	755
C2.4.257	VCVT (floating-point to integer)	757
C2.4.258	VCVT (integer to floating-point)	759
C2.4.259	VCVTA	761
C2.4.260	VCVTB	762
C2.4.261	VCVTM	764
C2.4.262	VCVTN	765
C2.4.263	VCVTP	766
C2.4.264	VCVTR	767
C2.4.265	VCVTT	769
C2.4.266	VDIV	771
C2.4.267	VFMA	772
C2.4.268	VFMS	773
C2.4.269	VFNMA	774
C2.4.270	VFNMS	776
C2.4.271	VLDM	778
C2.4.272	VLDR	781
C2.4.273	VLLDM	783
C2.4.274	VLSTM	785
C2.4.275	VMAXNM	787
C2.4.276	VMINNM	789
C2.4.277	VMLA	791
C2.4.278	VMLS	792
C2.4.279	VMOV (between general-purpose register and single-precision register)	794
C2.4.280	VMOV (between two general-purpose registers and a doubleword register)	795
C2.4.281	VMOV (between two general-purpose registers and two single-precision registers)	797
C2.4.282	VMOV (half of doubleword register to single general-purpose register)	799
C2.4.283	VMOV (immediate)	800
C2.4.284	VMOV (register)	801
C2.4.285	VMOV (single general-purpose register to half of doubleword register)	802
C2.4.286	VMRS	803
C2.4.287	VMSR	804
C2.4.288	VMUL	805
C2.4.289	VNEG	806
C2.4.290	VNMLA	807
C2.4.291	VNMLS	809
C2.4.292	VNMUL	811
C2.4.293	VPOP	813
C2.4.294	VPUSH	815
C2.4.295	VRINTA	817
C2.4.296	VRINTM	819

C2.4.297	VRINTN	821
C2.4.298	VRINTP	823
C2.4.299	VRINTR	825
C2.4.300	VRINTX	826
C2.4.301	VRINTZ	827
C2.4.302	VSEL	828
C2.4.303	VSQRT	830
C2.4.304	VSTM	831
C2.4.305	VSTR	834
C2.4.306	VSUB	836
C2.4.307	WFE	837
C2.4.308	WFI	838
C2.4.309	YIELD	839

Part D Armv8-M Registers

Chapter D1

Register Specification

D1.1	Register index	842
D1.1.1	Special and general-purpose registers	842
D1.1.2	Payloads	843
D1.1.3	Instrumentation Macrocell	843
D1.1.4	Data Watchpoint and Trace	843
D1.1.5	Flash Patch and Breakpoint	844
D1.1.6	Implementation Control Block	844
D1.1.7	SysTick Timer	844
D1.1.8	Nested Vectored Interrupt Controller	845
D1.1.9	System Control Block	845
D1.1.10	Memory Protection Unit	846
D1.1.11	Security Attribution Unit	846
D1.1.12	Debug Control Block	846
D1.1.13	Software Interrupt Generation	847
D1.1.14	Floating-Point Extension	847
D1.1.15	Cache Maintenance Operations	847
D1.1.16	Debug Identification Block	847
D1.1.17	Implementation Control Block (NS alias)	848
D1.1.18	SysTick Timer (NS alias)	848
D1.1.19	Nested Vectored Interrupt Controller (NS alias)	848
D1.1.20	System Control Block (NS alias)	848
D1.1.21	Memory Protection Unit (NS alias)	849
D1.1.22	Debug Control Block (NS alias)	849
D1.1.23	Software Interrupt Generation (NS alias)	850
D1.1.24	Floating-Point Extension (NS alias)	850
D1.1.25	Cache Maintenance Operations (NS alias)	850
D1.1.26	Debug Identification Block (NS alias)	850
D1.1.27	Trace Port Interface Unit	851
D1.2	Alphabetical list of registers	852
D1.2.1	ACTLR, Auxiliary Control Register	853
D1.2.2	AFSR, Auxiliary Fault Status Register	854
D1.2.3	AIRCR, Application Interrupt and Reset Control Register	855
D1.2.4	APSR, Application Program Status Register	859
D1.2.5	BASEPRI, Base Priority Mask Register	861
D1.2.6	BFAR, BusFault Address Register	862
D1.2.7	BFSR, BusFault Status Register	863
D1.2.8	BPIALL, Branch Predictor Invalidate All	866

D1.2.9	CCR, Configuration and Control Register	867
D1.2.10	CCSIDR, Current Cache Size ID register	871
D1.2.11	CFSR, Configurable Fault Status Register	873
D1.2.12	CLIDR, Cache Level ID Register	874
D1.2.13	CONTROL, Control Register	876
D1.2.14	CPACR, Coprocessor Access Control Register	878
D1.2.15	CPPWR, Coprocessor Power Control Register	880
D1.2.16	CPUID, CPUID Base Register	883
D1.2.17	CSSELR, Cache Size Selection Register	885
D1.2.18	CTR, Cache Type Register	887
D1.2.19	DAUTHCTRL, Debug Authentication Control Register	889
D1.2.20	DAUTHSTATUS, Debug Authentication Status Register	891
D1.2.21	DCCIMVAC, Data Cache line Clean and Invalidate by Address to PoC	893
D1.2.22	DCCISW, Data Cache line Clean and Invalidate by Set/Way	894
D1.2.23	DCCMVAC, Data Cache line Clean by Address to PoC	895
D1.2.24	DCCMVAU, Data Cache line Clean by address to PoU	896
D1.2.25	DCCSW, Data Cache Clean line by Set/Way	897
D1.2.26	DCIDR0, SCS Component Identification Register 0	898
D1.2.27	DCIDR1, SCS Component Identification Register 1	899
D1.2.28	DCIDR2, SCS Component Identification Register 2	900
D1.2.29	DCIDR3, SCS Component Identification Register 3	901
D1.2.30	DCIMVAC, Data Cache line Invalidate by Address to PoC	902
D1.2.31	DCISW, Data Cache line Invalidate by Set/Way	903
D1.2.32	DCRDR, Debug Core Register Data Register	904
D1.2.33	DCRSR, Debug Core Register Select Register	905
D1.2.34	DDEVARCH, SCS Device Architecture Register	908
D1.2.35	DDEVTYPE, SCS Device Type Register	910
D1.2.36	DEMCR, Debug Exception and Monitor Control Register	911
D1.2.37	DFSR, Debug Fault Status Register	917
D1.2.38	DHCSR, Debug Halting Control and Status Register	919
D1.2.39	DLAR, SCS Software Lock Access Register	925
D1.2.40	DLSR, SCS Software Lock Status Register	926
D1.2.41	DPIDR0, SCS Peripheral Identification Register 0	928
D1.2.42	DPIDR1, SCS Peripheral Identification Register 1	929
D1.2.43	DPIDR2, SCS Peripheral Identification Register 2	930
D1.2.44	DPIDR3, SCS Peripheral Identification Register 3	931
D1.2.45	DPIDR4, SCS Peripheral Identification Register 4	932
D1.2.46	DPIDR5, SCS Peripheral Identification Register 5	933
D1.2.47	DPIDR6, SCS Peripheral Identification Register 6	934
D1.2.48	DPIDR7, SCS Peripheral Identification Register 7	935
D1.2.49	DSCSR, Debug Security Control and Status Register	936
D1.2.50	DWT_CIDR0, DWT Component Identification Register 0	938
D1.2.51	DWT_CIDR1, DWT Component Identification Register 1	939
D1.2.52	DWT_CIDR2, DWT Component Identification Register 2	940
D1.2.53	DWT_CIDR3, DWT Component Identification Register 3	941
D1.2.54	DWT_COMPn, DWT Comparator Register, n = 0 - 14	942
D1.2.55	DWT_CPICNT, DWT CPI Count Register	944
D1.2.56	DWT_CTRL, DWT Control Register	945
D1.2.57	DWT_CYCCNT, DWT Cycle Count Register	950
D1.2.58	DWT_DEVARCH, DWT Device Architecture Register	951
D1.2.59	DWT_DEVTYPE, DWT Device Type Register	953
D1.2.60	DWT_EXCCNT, DWT Exception Overhead Count Register	954
D1.2.61	DWT_FOLDCNT, DWT Folded Instruction Count Register	955
D1.2.62	DWT_FUNCTIONn, DWT Comparator Function Register, n = 0 - 14	956
D1.2.63	DWT_LAR, DWT Software Lock Access Register	961

D1.2.64	DWT_LSR, DWT Software Lock Status Register	962
D1.2.65	DWT_LSUCNT, DWT LSU Count Register	964
D1.2.66	DWT_PCSR, DWT Program Counter Sample Register	965
D1.2.67	DWT_PIDR0, DWT Peripheral Identification Register 0	966
D1.2.68	DWT_PIDR1, DWT Peripheral Identification Register 1	967
D1.2.69	DWT_PIDR2, DWT Peripheral Identification Register 2	968
D1.2.70	DWT_PIDR3, DWT Peripheral Identification Register 3	969
D1.2.71	DWT_PIDR4, DWT Peripheral Identification Register 4	970
D1.2.72	DWT_PIDR5, DWT Peripheral Identification Register 5	971
D1.2.73	DWT_PIDR6, DWT Peripheral Identification Register 6	972
D1.2.74	DWT_PIDR7, DWT Peripheral Identification Register 7	973
D1.2.75	DWT_SLEEPCNT, DWT Sleep Count Register	974
D1.2.76	EPSR, Execution Program Status Register	976
D1.2.77	EXC_RETURN, Exception Return Payload	977
D1.2.78	FAULTMASK, Fault Mask Register	979
D1.2.79	FNC_RETURN, Function Return Payload	980
D1.2.80	FPCAR, Floating-Point Context Address Register	981
D1.2.81	FPCCR, Floating-Point Context Control Register	982
D1.2.82	FPDSCR, Floating-Point Default Status Control Register	987
D1.2.83	FPSCR, Floating-point Status and Control Register	988
D1.2.84	FP_CIDR0, FP Component Identification Register 0	992
D1.2.85	FP_CIDR1, FP Component Identification Register 1	993
D1.2.86	FP_CIDR2, FP Component Identification Register 2	994
D1.2.87	FP_CIDR3, FP Component Identification Register 3	995
D1.2.88	FP_COMPn, Flash Patch Comparator Register, n = 0 - 125	996
D1.2.89	FP_CTRL, Flash Patch Control Register	997
D1.2.90	FP_DEVARCH, FPB Device Architecture Register	999
D1.2.91	FP_DEVTYPE, FPB Device Type Register	1001
D1.2.92	FP_LAR, FPB Software Lock Access Register	1002
D1.2.93	FP_LSR, FPB Software Lock Status Register	1003
D1.2.94	FP_PIDR0, FP Peripheral Identification Register 0	1005
D1.2.95	FP_PIDR1, FP Peripheral Identification Register 1	1006
D1.2.96	FP_PIDR2, FP Peripheral Identification Register 2	1007
D1.2.97	FP_PIDR3, FP Peripheral Identification Register 3	1008
D1.2.98	FP_PIDR4, FP Peripheral Identification Register 4	1009
D1.2.99	FP_PIDR5, FP Peripheral Identification Register 5	1010
D1.2.100	FP_PIDR6, FP Peripheral Identification Register 6	1011
D1.2.101	FP_PIDR7, FP Peripheral Identification Register 7	1012
D1.2.102	FP_REMAP, Flash Patch Remap Register	1013
D1.2.103	HFSR, HardFault Status Register	1014
D1.2.104	ICIALLU, Instruction Cache Invalidate All to PoU	1016
D1.2.105	ICIMVAU, Instruction Cache line Invalidate by Address to PoU	1017
D1.2.106	ICSR, Interrupt Control and State Register	1018
D1.2.107	ICTR, Interrupt Controller Type Register	1024
D1.2.108	ID_AFR0, Auxiliary Feature Register 0	1025
D1.2.109	ID_DFR0, Debug Feature Register 0	1026
D1.2.110	ID_ISAR0, Instruction Set Attribute Register 0	1027
D1.2.111	ID_ISAR1, Instruction Set Attribute Register 1	1029
D1.2.112	ID_ISAR2, Instruction Set Attribute Register 2	1031
D1.2.113	ID_ISAR3, Instruction Set Attribute Register 3	1034
D1.2.114	ID_ISAR4, Instruction Set Attribute Register 4	1037
D1.2.115	ID_ISAR5, Instruction Set Attribute Register 5	1039
D1.2.116	ID_MMFR0, Memory Model Feature Register 0	1040
D1.2.117	ID_MMFR1, Memory Model Feature Register 1	1042
D1.2.118	ID_MMFR2, Memory Model Feature Register 2	1043

D1.2.119 ID_MMFR3, Memory Model Feature Register 3 1044

D1.2.120 ID_PFR0, Processor Feature Register 0 1046

D1.2.121 ID_PFR1, Processor Feature Register 1 1047

D1.2.122 IPSR, Interrupt Program Status Register 1048

D1.2.123 ITM_CIDR0, ITM Component Identification Register 0 1049

D1.2.124 ITM_CIDR1, ITM Component Identification Register 1 1050

D1.2.125 ITM_CIDR2, ITM Component Identification Register 2 1051

D1.2.126 ITM_CIDR3, ITM Component Identification Register 3 1052

D1.2.127 ITM_DEVARCH, ITM Device Architecture Register 1053

D1.2.128 ITM_DEVTYPE, ITM Device Type Register 1055

D1.2.129 ITM_LAR, ITM Software Lock Access Register 1057

D1.2.130 ITM_LSR, ITM Software Lock Status Register 1058

D1.2.131 ITM_PIDR0, ITM Peripheral Identification Register 0 1060

D1.2.132 ITM_PIDR1, ITM Peripheral Identification Register 1 1061

D1.2.133 ITM_PIDR2, ITM Peripheral Identification Register 2 1062

D1.2.134 ITM_PIDR3, ITM Peripheral Identification Register 3 1063

D1.2.135 ITM_PIDR4, ITM Peripheral Identification Register 4 1064

D1.2.136 ITM_PIDR5, ITM Peripheral Identification Register 5 1065

D1.2.137 ITM_PIDR6, ITM Peripheral Identification Register 6 1066

D1.2.138 ITM_PIDR7, ITM Peripheral Identification Register 7 1067

D1.2.139 ITM_STIMn, ITM Stimulus Port Register, n = 0 - 255 1068

D1.2.140 ITM_TCR, ITM Trace Control Register 1070

D1.2.141 ITM_TERN, ITM Trace Enable Register, n = 0 - 7 1074

D1.2.142 ITM_TPR, ITM Trace Privilege Register 1075

D1.2.143 LR, Link Register 1076

D1.2.144 MAIR_ATTR, Memory Attribute Indirection Register Attributes 1077

D1.2.145 MMFAR, MemManage Fault Address Register 1079

D1.2.146 MMFSR, MemManage Fault Status Register 1080

D1.2.147 MPU_CTRL, MPU Control Register 1083

D1.2.148 MPU_MAIR0, MPU Memory Attribute Indirection Register 0 1085

D1.2.149 MPU_MAIR1, MPU Memory Attribute Indirection Register 1 1086

D1.2.150 MPU_RBAR, MPU Region Base Address Register 1087

D1.2.151 MPU_RBAR_An, MPU Region Base Address Register Alias, n = 1 - 3 1089

D1.2.152 MPU_RLAR, MPU Region Limit Address Register 1091

D1.2.153 MPU_RLAR_An, MPU Region Limit Address Register Alias, n = 1 - 3 1093

D1.2.154 MPU_RNR, MPU Region Number Register 1095

D1.2.155 MPU_TYPE, MPU Type Register 1096

D1.2.156 MSPLIM, Main Stack Pointer Limit Register 1097

D1.2.157 MVFR0, Media and VFP Feature Register 0 1098

D1.2.158 MVFR1, Media and VFP Feature Register 1 1100

D1.2.159 MVFR2, Media and VFP Feature Register 2 1102

D1.2.160 NSACR, Non-secure Access Control Register 1103

D1.2.161 NVIC_IABRn, Interrupt Active Bit Register, n = 0 - 15 1105

D1.2.162 NVIC_ICERn, Interrupt Clear Enable Register, n = 0 - 15 1106

D1.2.163 NVIC_ICPRn, Interrupt Clear Pending Register, n = 0 - 15 1107

D1.2.164 NVIC_IPRn, Interrupt Priority Register, n = 0 - 123 1108

D1.2.165 NVIC_ISERN, Interrupt Set Enable Register, n = 0 - 15 1109

D1.2.166 NVIC_ISPRn, Interrupt Set Pending Register, n = 0 - 15 1110

D1.2.167 NVIC_ITNSn, Interrupt Target Non-secure Register, n = 0 - 15 1112

D1.2.168 PC, Program Counter 1113

D1.2.169 PRIMASK, Exception Mask Register 1114

D1.2.170 PSPLIM, Process Stack Pointer Limit Register 1115

D1.2.171 Rn, General-Purpose Register, n = 0 - 12 1116

D1.2.172 RETPSR, Combined Exception Return Program Status Registers 1117

D1.2.173 SAU_CTRL, SAU Control Register 1119

D1.2.174 SAU_RBAR, SAU Region Base Address Register	1121
D1.2.175 SAU_RLAR, SAU Region Limit Address Register	1122
D1.2.176 SAU_RNR, SAU Region Number Register	1124
D1.2.177 SAU_TYPE, SAU Type Register	1125
D1.2.178 SCR, System Control Register	1126
D1.2.179 SFAR, Secure Fault Address Register	1128
D1.2.180 SFSR, Secure Fault Status Register	1129
D1.2.181 SHCSR, System Handler Control and State Register	1132
D1.2.182 SHPR1, System Handler Priority Register 1	1139
D1.2.183 SHPR2, System Handler Priority Register 2	1141
D1.2.184 SHPR3, System Handler Priority Register 3	1142
D1.2.185 SP, Current Stack Pointer Register	1144
D1.2.186 SP_NS, Stack Pointer (Non-secure)	1145
D1.2.187 STIR, Software Triggered Interrupt Register	1146
D1.2.188 SYST_CALIB, SysTick Calibration Value Register	1147
D1.2.189 SYST_CSR, SysTick Control and Status Register	1149
D1.2.190 SYST_CVR, SysTick Current Value Register	1152
D1.2.191 SYST_RVR, SysTick Reload Value Register	1153
D1.2.192 TPIU_ACP, TPIU Asynchronous Clock Prescaler Register	1154
D1.2.193 TPIU_CIDR0, TPIU Component Identification Register 0	1155
D1.2.194 TPIU_CIDR1, TPIU Component Identification Register 1	1156
D1.2.195 TPIU_CIDR2, TPIU Component Identification Register 2	1157
D1.2.196 TPIU_CIDR3, TPIU Component Identification Register 3	1158
D1.2.197 TPIU_CSPSR, TPIU Current Parallel Port Sizes Register	1159
D1.2.198 TPIU_DEVTYP, TPIU Device Type Register	1160
D1.2.199 TPIU_FFCR, TPIU Formatter and Flush Control Register	1162
D1.2.200 TPIU_FFSR, TPIU Formatter and Flush Status Register	1164
D1.2.201 TPIU_LAR, TPIU Software Lock Access Register	1166
D1.2.202 TPIU_LSR, TPIU Software Lock Status Register	1167
D1.2.203 TPIU_PIDR0, TPIU Peripheral Identification Register 0	1169
D1.2.204 TPIU_PIDR1, TPIU Peripheral Identification Register 1	1170
D1.2.205 TPIU_PIDR2, TPIU Peripheral Identification Register 2	1171
D1.2.206 TPIU_PIDR3, TPIU Peripheral Identification Register 3	1172
D1.2.207 TPIU_PIDR4, TPIU Peripheral Identification Register 4	1173
D1.2.208 TPIU_PIDR5, TPIU Peripheral Identification Register 5	1174
D1.2.209 TPIU_PIDR6, TPIU Peripheral Identification Register 6	1175
D1.2.210 TPIU_PIDR7, TPIU Peripheral Identification Register 7	1176
D1.2.211 TPIU_PSCR, TPIU Periodic Synchronization Control Register	1177
D1.2.212 TPIU_SPPR, TPIU Selected Pin Protocol Register	1179
D1.2.213 TPIU_SSPSR, TPIU Supported Parallel Port Sizes Register	1180
D1.2.214 TPIU_TYPE, TPIU Device Identifier Register	1181
D1.2.215 TT_RESP, Test Target Response Payload	1183
D1.2.216 UFSR, UsageFault Status Register	1186
D1.2.217 VTOR, Vector Table Offset Register	1189
D1.2.218 XPSR, Combined Program Status Registers	1190

Part E Armv8-M Pseudocode

Chapter E1

Arm Pseudocode Definition

E1.1 About the Arm pseudocode	1194
E1.1.1 General limitations of Arm pseudocode	1194
E1.2 Data types	1195
E1.2.1 General data type rules	1195
E1.2.2 Bitstrings	1195

E1.2.3	Integers	1196
E1.2.4	Reals	1196
E1.2.5	Booleans	1197
E1.2.6	Enumerations	1197
E1.2.7	Structures	1198
E1.2.8	Tuples	1199
E1.2.9	Arrays	1199
E1.3	Operators	1201
E1.3.1	Relational operators	1201
E1.3.2	Boolean operators	1201
E1.3.3	Bitstring operators	1202
E1.3.4	Arithmetic operators	1203
E1.3.5	The assignment operator	1204
E1.3.6	Precedence rules	1205
E1.3.7	Conditional expressions	1205
E1.3.8	Operator polymorphism	1205
E1.4	Statements and control structures	1207
E1.4.1	Statements and Indentation	1207
E1.4.2	Function and procedure calls	1207
E1.4.3	Conditional control structures	1208
E1.4.4	Loop control structures	1209
E1.4.5	Special statements	1210
E1.4.6	Comments	1211
E1.5	Built-in functions	1212
E1.5.1	Bitstring manipulation functions	1212
E1.5.2	Arithmetic functions	1213
E1.6	Arm pseudocode definition index	1215
E1.7	Additional functions	1218
E1.7.1	IsSee()	1218
E1.7.2	IsUndefined()	1218

Chapter E2

Pseudocode Specification

E2.1	Alphabetical Pseudocode List	1220
E2.1.1	_D	1220
E2.1.2	_ITStateChanged	1220
E2.1.3	_Mem	1220
E2.1.4	_NextInstrAddr	1220
E2.1.5	_NextInstrITState	1220
E2.1.6	_PCChanged	1220
E2.1.7	_PendingReturnOperation	1220
E2.1.8	_R	1220
E2.1.9	_SP	1221
E2.1.10	Abs	1221
E2.1.11	AccessAttributes	1221
E2.1.12	AccType	1222
E2.1.13	ActivateException	1222
E2.1.14	AddressDescriptor	1222
E2.1.15	AddWithCarry	1222
E2.1.16	Align	1223
E2.1.17	ALUWritePC	1223
E2.1.18	ASR	1223
E2.1.19	ASR_C	1223
E2.1.20	BigEndian	1223
E2.1.21	BigEndianReverse	1223
E2.1.22	BitCount	1224

Contents

E2.1.23	BKPTInstrDebugEvent	1224
E2.1.24	BLXWritePC	1224
E2.1.25	BranchTo	1224
E2.1.26	BranchToAndCommit	1225
E2.1.27	BranchToNS	1225
E2.1.28	BranchWritePC	1225
E2.1.29	BXWritePC	1225
E2.1.30	CallSupervisor	1226
E2.1.31	CanHaltOnEvent	1226
E2.1.32	CanPendMonitorOnEvent	1226
E2.1.33	CheckCPEnabled	1226
E2.1.34	CheckDecodeFaults	1227
E2.1.35	CheckPermission	1227
E2.1.36	ClearEventRegister	1228
E2.1.37	ClearExclusiveByAddress	1228
E2.1.38	ClearExclusiveLocal	1228
E2.1.39	ComparePriorities	1228
E2.1.40	ConditionHolds	1229
E2.1.41	ConditionPassed	1229
E2.1.42	ConstrainUnpredictableBool	1229
E2.1.43	ConsumeExcStackFrame	1229
E2.1.44	ConsumptionOfSpeculativeDataBarrier	1230
E2.1.45	Coproc_Accepted	1230
E2.1.46	Coproc_DoneLoading	1230
E2.1.47	Coproc_DoneStoring	1230
E2.1.48	Coproc_GetOneWord	1230
E2.1.49	Coproc_GetTwoWords	1230
E2.1.50	Coproc_GetWordToStore	1231
E2.1.51	Coproc_InternalOperation	1231
E2.1.52	Coproc_SendLoadedWord	1231
E2.1.53	Coproc_SendOneWord	1231
E2.1.54	Coproc_SendTwoWords	1231
E2.1.55	CountLeadingSignBits	1231
E2.1.56	CountLeadingZeroBits	1231
E2.1.57	CreateException	1231
E2.1.58	CurrentCond	1232
E2.1.59	CurrentMode	1232
E2.1.60	CurrentModelsPrivileged	1233
E2.1.61	D	1233
E2.1.62	DataMemoryBarrier	1233
E2.1.63	DataSynchronizationBarrier	1233
E2.1.64	DeActivate	1233
E2.1.65	Debug_authentication	1234
E2.1.66	DecodeExecute	1234
E2.1.67	DecodeImmShift	1234
E2.1.68	DecodeRegShift	1234
E2.1.69	DefaultExclInfo	1235
E2.1.70	DefaultMemoryAttributes	1235
E2.1.71	DefaultPermissions	1236
E2.1.72	DerivedLateArrival	1236
E2.1.73	DeviceType	1238
E2.1.74	DWT_AddressCompare	1238
E2.1.75	DWT_CycCountMatch	1238
E2.1.76	DWT_DataAddressMatch	1238
E2.1.77	DWT_DataMatch	1239

E2.1.78	DWT_DataValueMatch	1240
E2.1.79	DWT_InstructionAddressMatch	1241
E2.1.80	DWT_InstructionMatch	1242
E2.1.81	DWT_ValidMatch	1242
E2.1.82	EndOfInstruction	1243
E2.1.83	EventRegistered	1243
E2.1.84	ExceptionActiveBitCount	1243
E2.1.85	ExceptionDetails	1243
E2.1.86	ExceptionEnabled	1244
E2.1.87	ExceptionEntry	1244
E2.1.88	ExceptionPriority	1245
E2.1.89	ExceptionReturn	1246
E2.1.90	ExceptionTaken	1247
E2.1.91	ExceptionTargetsSecure	1248
E2.1.92	ExclInfo	1249
E2.1.93	ExclusiveMonitorsPass	1249
E2.1.94	ExecuteCPCheck	1250
E2.1.95	ExecuteFPCheck	1250
E2.1.96	ExecutionPriority	1250
E2.1.97	ExternalInvasiveDebugEnabled	1251
E2.1.98	ExternalNoninvasiveDebugEnabled	1251
E2.1.99	ExternalSecureInvasiveDebugEnabled	1252
E2.1.100	ExternalSecureNoninvasiveDebugEnabled	1252
E2.1.101	ExternalSecureSelfHostedDebugEnabled	1252
E2.1.102	FaultNumbers	1252
E2.1.103	FetchInstr	1253
E2.1.104	FindPriv	1254
E2.1.105	FixedToFP	1254
E2.1.106	FPAbs	1254
E2.1.107	FPAdd	1254
E2.1.108	FPB_BreakpointMatch	1255
E2.1.109	FPB_CheckBreakPoint	1255
E2.1.110	FPB_CheckMatchAddress	1255
E2.1.111	FPCompare	1255
E2.1.112	FPDefaultNaN	1256
E2.1.113	FPDiv	1256
E2.1.114	FPDoubleToHalf	1256
E2.1.115	FPDoubleToSingle	1257
E2.1.116	FPExc	1257
E2.1.117	FPHalfToDouble	1257
E2.1.118	FPHalfToSingle	1258
E2.1.119	FPInfinity	1258
E2.1.120	FPMax	1258
E2.1.121	FPMaxNormal	1259
E2.1.122	FPMaxNum	1259
E2.1.123	FPMin	1259
E2.1.124	FPMinNum	1260
E2.1.125	FPMul	1260
E2.1.126	FPMulAdd	1260
E2.1.127	FPNeg	1261
E2.1.128	FPProcessException	1261
E2.1.129	FPProcessNaN	1262
E2.1.130	FPProcessNaNs	1262
E2.1.131	FPProcessNaNs3	1262
E2.1.132	FPRound	1263

Contents

E2.1.133	FPRoundInt	1264
E2.1.134	FPSingleToDouble	1265
E2.1.135	FPSingleToHalf	1265
E2.1.136	FPSqrt	1266
E2.1.137	FPSub	1266
E2.1.138	FPToFixed	1267
E2.1.139	FPToFixedDirected	1267
E2.1.140	FPType	1268
E2.1.141	FPUnpack	1268
E2.1.142	FPZero	1269
E2.1.143	FunctionReturn	1270
E2.1.144	GenerateCoproprocessorException	1270
E2.1.145	GenerateDebugEventResponse	1271
E2.1.146	GenerateIntegerZeroDivide	1271
E2.1.147	HaltingDebugAllowed	1271
E2.1.148	HandleException	1271
E2.1.149	HaveDebugMonitor	1272
E2.1.150	HaveDSPExt	1272
E2.1.151	HaveDWT	1272
E2.1.152	HaveFPB	1272
E2.1.153	HaveFPExt	1272
E2.1.154	HaveHaltingDebug	1272
E2.1.155	HaveITM	1272
E2.1.156	HaveMainExt	1272
E2.1.157	HaveSecurityExt	1273
E2.1.158	HaveSPFPOnly	1273
E2.1.159	HaveSysTick	1273
E2.1.160	HighestPri	1273
E2.1.161	HighestSetBit	1273
E2.1.162	Hint_Debug	1273
E2.1.163	Hint_PreloadData	1273
E2.1.164	Hint_PreloadDataForWrite	1274
E2.1.165	Hint_PreloadInstr	1274
E2.1.166	Hint_Yield	1274
E2.1.167	IDAUCheck	1274
E2.1.168	InITBlock	1274
E2.1.169	InstructionAdvance	1274
E2.1.170	InstructionSynchronizationBarrier	1276
E2.1.171	Int	1276
E2.1.172	IntegerZeroDivideTrappingEnabled	1276
E2.1.173	IsAccessible	1276
E2.1.174	IsActiveForState	1277
E2.1.175	IsAligned	1277
E2.1.176	IsCPEEnabled	1277
E2.1.177	IsCPInstruction	1278
E2.1.178	IsDWTConfigUnpredictable	1278
E2.1.179	IsDWTEnabled	1279
E2.1.180	IsExceptionTargetConfigurable	1279
E2.1.181	IsExclusiveGlobal	1280
E2.1.182	IsExclusiveLocal	1280
E2.1.183	IsIrqValid	1280
E2.1.184	IsOnes	1280
E2.1.185	IsReqExcPriNeg	1280
E2.1.186	IsSecure	1281
E2.1.187	IsZero	1281

Contents

E2.1.188	IsZeroBit	1281
E2.1.189	ITAdvance	1281
E2.1.190	ITSTATE	1281
E2.1.191	ITSTATEType	1281
E2.1.192	LastInITBlock	1282
E2.1.193	LoadWritePC	1282
E2.1.194	Lockup	1282
E2.1.195	LookupRName	1282
E2.1.196	LookupSP	1283
E2.1.197	LookupSP_with_security_mode	1283
E2.1.198	LookupSPLim	1283
E2.1.199	LowestSetBit	1284
E2.1.200	LR	1284
E2.1.201	LSL	1284
E2.1.202	LSL_C	1284
E2.1.203	LSR	1284
E2.1.204	LSR_C	1285
E2.1.205	MAIRDecode	1285
E2.1.206	MarkExclusiveGlobal	1286
E2.1.207	MarkExclusiveLocal	1286
E2.1.208	Max	1286
E2.1.209	MaxExceptionNum	1286
E2.1.210	MemA	1287
E2.1.211	MemA_with_priv	1287
E2.1.212	MemA_with_priv_security	1287
E2.1.213	MemI	1289
E2.1.214	MemO	1289
E2.1.215	MemoryAttributes	1290
E2.1.216	MemType	1290
E2.1.217	MemU	1290
E2.1.218	MemU_unpriv	1290
E2.1.219	MemU_with_priv	1291
E2.1.220	MergeExclInfo	1291
E2.1.221	Min	1292
E2.1.222	MPUCheck	1292
E2.1.223	NextInstrAddr	1294
E2.1.224	NextInstrITState	1294
E2.1.225	NoninvasiveDebugAllowed	1294
E2.1.226	Ones	1294
E2.1.227	PC	1295
E2.1.228	PEMode	1295
E2.1.229	PendingExceptionDetails	1295
E2.1.230	PendReturnOperation	1295
E2.1.231	Permissions	1295
E2.1.232	PopStack	1295
E2.1.233	PreserveFPState	1298
E2.1.234	ProcessorID	1299
E2.1.235	PushCalleeStack	1299
E2.1.236	PushStack	1300
E2.1.237	R	1301
E2.1.238	RaiseAsyncBusFault	1302
E2.1.239	RawExecutionPriority	1302
E2.1.240	Replicate	1303
E2.1.241	ResetSCSRegs	1303
E2.1.242	RestrictedNSPri	1303

Contents

E2.1.243	ReturnState	1303
E2.1.244	RName	1304
E2.1.245	ROR	1304
E2.1.246	ROR_C	1304
E2.1.247	RoundDown	1304
E2.1.248	RoundTowardsZero	1304
E2.1.249	RoundUp	1305
E2.1.250	RRX	1305
E2.1.251	RRX_C	1305
E2.1.252	RSPCheck	1305
E2.1.253	S	1305
E2.1.254	Sat	1306
E2.1.255	SatQ	1306
E2.1.256	SAttributes	1306
E2.1.257	SCS_UpdateStatusRegs	1306
E2.1.258	SecureDebugMonitorAllowed	1306
E2.1.259	SecureHaltingDebugAllowed	1306
E2.1.260	SecureNoninvasiveDebugAllowed	1307
E2.1.261	SecurityCheck	1307
E2.1.262	SecurityState	1308
E2.1.263	SendEvent	1308
E2.1.264	SerializeVFP	1308
E2.1.265	SetActive	1308
E2.1.266	SetDWTDebugEvent	1309
E2.1.267	SetEventRegister	1309
E2.1.268	SetExclusiveMonitors	1309
E2.1.269	SetITSTATEAndCommit	1310
E2.1.270	SetMonStep	1310
E2.1.271	SetPending	1310
E2.1.272	SetThisInstrDetails	1310
E2.1.273	Shift	1310
E2.1.274	Shift_C	1311
E2.1.275	SignedSat	1311
E2.1.276	SignedSatQ	1311
E2.1.277	SignExtend	1311
E2.1.278	SleepOnExit	1312
E2.1.279	SP	1312
E2.1.280	SP_Main	1312
E2.1.281	SP_Main_NonSecure	1312
E2.1.282	SP_Main_Secure	1312
E2.1.283	SP_Process	1313
E2.1.284	SP_Process_NonSecure	1313
E2.1.285	SP_Process_Secure	1313
E2.1.286	SpeculativeSynchronizationBarrier	1313
E2.1.287	SRTYPE	1314
E2.1.288	Stack	1314
E2.1.289	StandardFPSCRValue	1314
E2.1.290	SteppingDebug	1315
E2.1.291	T32ExpandImm	1315
E2.1.292	T32ExpandImm_C	1315
E2.1.293	TailChain	1316
E2.1.294	TakePreserveFPEException	1316
E2.1.295	TakeReset	1317
E2.1.296	ThisInstr	1318
E2.1.297	ThisInstrAddr	1318

E2.1.298	ThisInstrITState	1319
E2.1.299	ThisInstrLength	1319
E2.1.300	TopLevel	1319
E2.1.301	TTResp	1320
E2.1.302	UnsignedSat	1321
E2.1.303	UnsignedSatQ	1321
E2.1.304	UpdateFPCCR	1321
E2.1.305	UpdateSecureDebugEnable	1322
E2.1.306	ValidateAddress	1323
E2.1.307	ValidateExceptionReturn	1324
E2.1.308	Vector	1325
E2.1.309	VFPExcBarrier	1325
E2.1.310	VFPExpandImm	1325
E2.1.311	VFPNegMul	1326
E2.1.312	VFPSmallRegisterBank	1326
E2.1.313	WaitForEvent	1326
E2.1.314	WaitForInterrupt	1326
E2.1.315	ZeroExtend	1326
E2.1.316	Zeros	1326

Part F Debug Packet Protocols

Chapter F1

ITM and DWT Packet Protocol Specification

F1.1	About the ITM and DWT packets	1329
F1.1.1	Uses of ITM and DWT packets	1329
F1.1.2	ITM and DWT protocol packet headers	1329
F1.1.3	Packet transmission by the trace sink	1329
F1.2	Alphabetical list of DWT and ITM packets	1331
F1.2.1	Data Trace Data Address packet	1331
F1.2.2	Data Trace Data Value packet	1332
F1.2.3	Data Trace Match packet	1334
F1.2.4	Data Trace PC Value packet	1335
F1.2.5	Event Counter packet	1337
F1.2.6	Exception Trace packet	1338
F1.2.7	Extension packet	1339
F1.2.8	Global Timestamp 1 packet	1341
F1.2.9	Global Timestamp 2 packet	1343
F1.2.10	Instrumentation packet	1345
F1.2.11	Local Timestamp 1 packet	1346
F1.2.12	Local Timestamp 2 packet	1348
F1.2.13	Overflow packet	1349
F1.2.14	Periodic PC Sample packet	1350
F1.2.15	Synchronization packet	1351

Glossary

Preface

This preface introduces the Armv8-M Architecture Reference Manual. It contains the following sections:

[About this book.](#)

[Using this book.](#)

[Conventions.](#)

[Additional reading.](#)

[Feedback.](#)

About this book

This manual documents the microcontroller profile of version 8 of the Arm Architecture, the Armv8-M architecture profile. For short definitions of all the Armv8 profiles, see [A1.2 About the Armv8 architecture, and architecture profiles](#).

This manual has the following parts:

Part A Provides an introduction to the Armv8-M architecture.

Part B Describes the architectural rules.

Part C Describes the T32 instruction set.

Part D Describes the registers.

Part E Describes the Armv8-M pseudocode.

Part F Describes the packet protocols.

Using this book

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

Part A, Armv8-M Architecture Introduction and Overview

Part A gives an overview of the Armv8-M architecture profile, including its relationship to the other Arm PE architectures. It introduces the terminology that describes the architecture, and gives an overview of the optional architectural extensions. It contains the following chapter:

[Chapter A1 *Introduction*](#)

Read this for an introduction to the Armv8-M architecture.

Part B, Armv8-M Architecture Rules

Part B describes the architecture rules. It contains the following chapters:

[Chapter B1 *Resets*](#)

Read this for a description of the reset rules.

[Chapter B2 *Power Management*](#)

Read this for a description of the power management rules.

[Chapter B3 *Programmers' Model*](#)

Read this for a description of the programmers model rules.

[Chapter B4 *Floating-point Support*](#)

Read this for a description of the floating-point support rules.

[Chapter B5 *Memory Model*](#)

Read this for a description of the memory model rules.

[Chapter B6 *The System Address Map*](#)

Read this for a description of the system address map rules.

[Chapter B7 *Synchronization and Semaphores*](#)

Read this for a description of the rules on non-blocking synchronization of shared memory.

[Chapter B8 *The Armv8-M Protected Memory System Architecture*](#)

Read this for a description of the protected memory system architecture rules.

[Chapter B9 *The System Timer, SysTick*](#)

Read this for a description of the system timer rules.

[Chapter B10 *Nested Vectored Interrupt Controller*](#)

Read this for a description of the *Nested Vectored Interrupt Controller* (NVIC) rules.

[Chapter B11 *Debug*](#)

Read this for a description of the debug rules.

[Chapter B12 *Debug and Trace Components*](#)

Read this for a description of the debug and trace component rules.

Part C, Armv8-M Instructions

Part C describes the instructions. It contains the following chapters:

[Chapter C1 *Instruction Set Overview*](#)

Read this for an overview of the instruction set and the instruction set encoding.

[Chapter C2 *Instruction Specification*](#)

Read this for a description of each instruction, arranged by instruction mnemonic.

Part D, Armv8-M Registers

Part D describes the registers. It contains the following chapter:

[Chapter D1 *Register Specification*](#)

Read this for a description of the registers.

Part E, Armv8-M Pseudocode

Part E describes the pseudocode. It contains the following chapters:

[Chapter E1 *Arm Pseudocode Definition*](#)

Read this for a definition of the pseudocode that Arm documentation uses.

[Chapter E2 *Pseudocode Specification*](#)

Read this for a description of the pseudocode.

Part F, Packet Protocols

Part F describes the packet protocols. It contains the following chapter:

[Chapter F1 *ITM and DWT Packet Protocol Specification*](#)

Read this for a description of the protocol for packets that are used to send the data generated by the ITM and DWT to an external debugger.

Conventions

The following sections describe conventions that this book can use:

[Typographical conventions.](#)

[Signals.](#)

[Numbers.](#)

[Pseudocode descriptions.](#)

[Assembler syntax descriptions.](#)

Typographical conventions

The typographical conventions are:

italic

Introduces special terminology, and denotes citations.

bold

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

`monospace`

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

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

SMALLCAPS

Used for a few terms that have specific technical meanings, and that are included in the Glossary.

Colored text Indicates a link. This can be:

- A URL, for example <https://developer.arm.com/>.
- A cross-reference, that includes the page number of the referenced information if it is not on the current page, for example, [Chapter B2 Power Management](#).
- A link, to a chapter or appendix, or to a glossary entry, or to the section of the document that defines the colored term, for example [tail-chaining](#).

Signals

In general this specification does not define processor signals, but it does include some signal examples and recommendations.

The signal conventions are:

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

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

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

Numbers

Numbers are normally written in decimal. Binary numbers are preceded by `0b`, and hexadecimal numbers by `0x`. In both cases, the prefix and the associated value are written in a `monospace` font, for example `0xFFFF0000`.

For both binary and hexadecimal numbers, where a bit is represented by the letter `x`, the value is irrelevant. For example a value expressed as `0b1x` can be either `0b11` or `0b10`.

To improve readability, long numbers can be written with an underscore separator between every four characters, for example `0xFFFF_0000_0000_0000`. Ignore any underscores when interpreting the value of a number.

Pseudocode descriptions

This book uses a form of pseudocode to provide precise descriptions of the specified functionality. This pseudocode is written in a `monospace` font, and is described in [Chapter E1 Arm Pseudocode Definition](#).

Assembler syntax descriptions

This book contains numerous syntax descriptions for assembler instructions and for components of assembler instructions. These are shown in a `monospace` font, and use the conventions described in [C1.2.5 Standard assembler syntax fields](#).

Additional reading

This section lists relevant publications from Arm and third parties.

See <https://developer.arm.com>, for access to Arm documentation.

Arm publications

- *Arm[®] Debug Interface Architecture Specification ADIV5.0 to ADIV5.2* (ARM IHI 0031).
- *Arm[®] Debug Interface Architecture Specification ADIV6.0* (ARM IHI 0074).
- *Arm[®] CoreSight[™] Architecture Specification* (ARM IHI 0029).
- *Arm[®] Embedded Trace Macrocell Architecture Specification ETMv4.0 to ETMv4.4* (ARM IHI 0064).
- *Embedded Trace Macrocell[®] ETMv1.0 to ETMv3.5 Architecture Specification* (ARM IHI 0014).
- *Arm[®] v6-M Architecture Reference Manual* (ARM DDI 0419).
- *Arm[®] v7-M Architecture Reference Manual* (ARM DDI 0403).
- *Arm[®] Architecture Reference Manual, Armv8, for Armv8-A architecture profile* (ARM DDI 0487).

Other publications

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

- ANSI/IEEE Std 754-1985 and ANSI/IEEE Std 754-2008, IEEE Standard for Binary Floating-Point Arithmetic. Unless otherwise indicated, references to IEEE 754 refer to either issue of the standard.

Note

This document does not adopt the terminology defined in the 2008 issue of the standard.

- JEP106, Standard Manufacturers Identification Code, JEDEC Solid State Technology Association.

Feedback

Arm welcomes feedback on its documentation.

Feedback on this book

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

- The title.
- The number, DDI0553A.m
- The page numbers to which your comments apply.
- The rule identifiers to which your comments apply, if applicable.
- A concise explanation of your comments.

Arm also welcomes general suggestions for additions and improvements.

Note

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

Part A
Armv8-M Architecture Introduction and Overview

Chapter A1

Introduction

This chapter introduces the Armv8 architecture, the architecture profiles it defines, and the Armv8-M architecture profile defined by this manual. It contains the following sections:

A1.1 Document layout and terminology on page 38.

A1.2 About the Armv8 architecture, and architecture profiles on page 41.

A1.3 The Armv8-M architecture profile on page 42.

A1.4 Armv8-M variants on page 44.

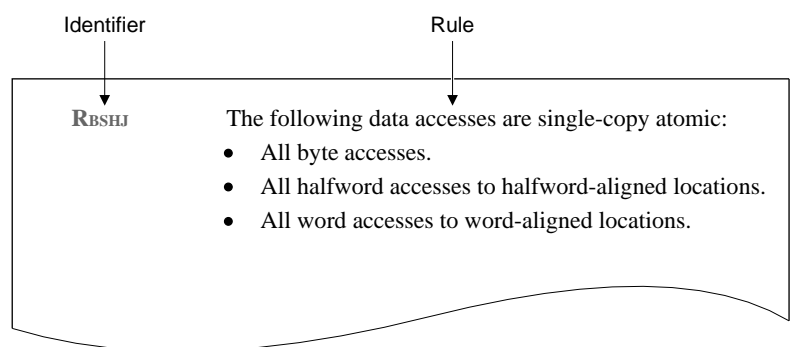
A1.1 Document layout and terminology

This section describes the structure and scope of this manual. This section also describes the terminology that this manual uses. It does not constitute part of the manual, and must not be interpreted as implementation guidance.

A1.1.1 Structure of the document

This architecture manual describes the behavior of the processing element as a set of individual rules.

Each rule is clearly identified by the letter R, followed by a random group of subscript letters that do not reflect any intended order or priority, for example R_{BSHJ}. In the following example, R_{BSHJ} is simply a random rule identifier that has no significance apart from uniquely identifying a rule in this manual.



Rules must not be read in isolation, and where more than one rule relating to a particular feature exists, individual rules are grouped into sections and subsections to provide the proper context. Where appropriate, these sections contain a short introduction to aid the reader.

An implementation that conforms to all the rules described in this specification constitutes an Armv8-M compliant implementation. An implementation whose behavior deviates from these rules is not compliant with the Armv8-M architecture.

Some sections contain additional information and guidance that do not constitute rules. This information and guidance is provided purely as an aid to understanding the architecture. Information statements are clearly identified by the letter I, followed by a random group of subscript letters, for example I_{PRTD}.

Note

Arm strongly recommends that implementers read *all* chapters and sections of this document to ensure that an implementation is compliant.

An implementation that conforms to all the rules described in this specification but chooses to ignore any additional information and guidance is compliant with the Armv8-M architecture.

In the following parts of this manual, architectural rules are not identified by a specific prefix and a random group of subscript letters:

- Parts of [Part C Armv8-M Instruction Set](#).
- [Part D Armv8-M Register Specification](#).
- [Part E Armv8-M Pseudocode](#).
- [Part F Armv8-M Debug Packet Protocols](#).

A1.1.2 Scope of the document

This manual contains only rules and information that relate specifically to the Armv8-M architecture. It does not include any information about other Arm architectures, nor does it describe similarities between Armv8-M and other architectures.

Readers must not assume that the rules provided in this specification are applicable to an Armv7-M or Armv6-M implementation, nor must they assume that the rules that are applicable to an Armv7-M or Armv6-M implementation are equally applicable to an Armv8-M implementation.

A1.1.3 Intended audience

This manual is written for users who want to design, implement, or program an Armv8-M PE in a range of Arm-compliant implementations from simple uniprocessor implementations to complex multiprocessor systems. It does not assume familiarity with previous versions of the M-Profile architecture.

The manual provides a precise, accurate, and correct set of rules that must be followed in order for an Armv8-M implementation to be architecturally compliant. It is an explicit reference manual, and not a general introduction to, or user guide for, the Armv8-M architecture.

A1.1.4 Terminology, phrases

This subsection identifies some standard words and phrases that are used in the Arm architecture documentation. These words and phrases have an Arm-specific definition, which is described in this section.

Architecturally visible

Something that is visible to the controlling agent. The controlling agent might be software.

Arm recommends

A particular usage that ensures consistency and usability. Following all the rules listed in this manual leads to a predictable outcome that is compliant with the architecture, but might produce an unexpected output. Adhering to a recommendation ensures that the output is as expected.

Arm strongly recommends

Something that is essentially mandatory, but that is outside the scope of the architecture described in this manual. Failing to adhere to a strong recommendation can break the system, although the PE itself remains compliant with the architecture that is described in this manual.

Finite time

An action will occur at some point in the future. Finite time does not make any statement about the time involved. However, delaying an action longer than is absolutely necessary might have an adverse impact on performance.

Permitted

Allowed behavior.

Required

Mandatory behavior.

Support

The implementation has implemented a particular feature.

A1.1.5 Terminology, Armv8-M specific terms

For definitions of Armv8-M specific terms, see the [Glossary](#).

A1.2 About the Armv8 architecture, and architecture profiles

Armv8-M is documented as one of a set of architecture profiles.

Arm defines three architecture profiles:

A Application profile:

- Supports a *Virtual Memory System Architecture* (VMSA) based on a *Memory Management Unit* (MMU).
- Supports the A64, A32, and T32 instruction sets.

R Real-time profile:

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

M Microcontroller profile, described in this manual:

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

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

A1.3 The Armv8-M architecture profile

The M-Profile architecture includes:

- The opportunity to include simple pipeline designs offering leading edge system performance levels in a broad range of markets and applications.
- Highly deterministic operation:
 - Single or low cycle count execution.
 - Minimal interrupt latency, with short pipelines.
 - Capable of cacheless operation.
- Excellent targeting of C/C++ code. This aligns with the Arm programming standards in this area:
 - Exception handlers are standard C/C++ functions, entered using standard calling conventions.
- Design support for deeply embedded systems:
 - Low pincount devices.
- Support for debug and software profiling for event-driven systems.

The simplest Armv8.0-M implementation, without any of the optional extensions, is a Baseline implementation, see [A1.4 Armv8-M variants on page 44](#). The Armv8.0-M Baseline offers improvements over previous M-Profile architectures in the following areas:

- The optional Security Extension.
- An improved, optional, *Memory Protection Unit* (MPU) model.
- Alignment with Armv8-A and Armv8-R memory types.
- Stack pointer limit checking.
- Improved support for multi-processing.
- Better alignment with C11 and C11++ standards.
- Enhanced debug capabilities.

A1.3.1 Security Extension

The Armv8-M architecture introduces a number of new instructions to the M-Profile architecture to support asset protection. These instructions are only available to implementations that support the Security Extension, see [A1.4 Armv8-M variants on page 44](#).

A1.3.2 MPU model

The Armv8-M architecture provides a default memory map and permits implementations to include an optional MPU. The optional MPU uses the Protected Memory System Architecture (PMSAv8) and contains improved flexibility in the MPU region definition, see [Chapter B8 The Armv8-M Protected Memory System Architecture on page 214](#).

A1.3.3 Nested Vector Interrupt Controller

The Nested Vector Interrupt Controller (NVIC) is used for integrated interrupt and exception handling and prioritization. Armv8-M increases the number of interrupts that can potentially be supported by the NVIC 480 for external sources, and includes automatic vectoring and priority management, and automatic state preservation. See [Chapter B10 Nested Vectored Interrupt Controller on page 226](#).

A1.3.4 Stack pointers

The Armv8-M architecture introduces stack limit registers that trigger an exception on a stack overflow. The number of stack limit registers available to an implementation is determined by the Armv8-M variant that is

implemented, see [B3.8 Stack pointer on page 63](#).

A1.3.5 The Armv8-M instruction set

Armv8-M only supports execution of T32 instructions. The Armv8-M architecture adds instructions to support:

- Improved facilitation of execute-only code generation.
- Improved code optimization.
- Exclusive memory access instructions to enhance support for multiprocessor systems.
- Semaphores and atomics (Load-Acquire/Store-Release instructions).

The optional *Floating-point Extension* adds floating-point instructions to the T32 instruction set, see [Chapter B4 Floating-point Support on page 126](#).

For more information about the instructions, see [Chapter C1 Instruction Set Overview on page 305](#) and Chapter C2, Instruction Specification.

A1.3.6 Debug

The Armv8-M architecture introduces:

- Enhanced breakpoint and watchpoint functionality.
- Improvements to the Instrumentation Trace Macrocell (ITM).
- Comprehensive trace and self-hosted debug extensions to make embedded software easier to debug and trace.

For more information about debug, see [Chapter B11 Debug on page 230](#) and [Chapter B12 Debug and Trace Components on page 268](#).

A1.4 Armv8-M variants

The Armv8-M architecture has the following optional extensions, which are abbreviated as follows:

DB - The Debug Extension

Note

For details about the individual features that constitute the Debug Extension, see [B11.1 Debug feature overview](#) on page 231.

DSP - The Digital Signal Processing Extension.

A PE that implements the DSP Extension must implement the Main Extension (M).

FP - The Floating-point Extension

A PE that implements the Floating-point Extension must implement the Main Extension (M).

The Floating-point Extension supports either single-precision floating-point instructions or both single-precision and double-precision floating-point instructions.

M - The Main Extension

A PE that implements the Main Extension implements the System Timer Extension.

Note

- A PE with the Main Extension is also referred to as a Mainline implementation.
- A PE without the Main Extension is also referred to as a Baseline implementation. A Baseline implementation has a subset of the instructions, registers, and features, of a Mainline implementation.
- Armv7-M compatibility requires the Main Extension.
- Armv6-M compatibility is provided by all Armv8-M implementations.

MPU - The Memory Protection Unit Extension

S - The Security Extension

Note

The Armv8-M Security Extension can also be referred to as Arm TrustZone for Armv8-M.

ST - The System Timer Extension

Where applicable, a line below each rule or information statement indicates the extensions that are required for the rule or information statement to apply, and any other notes.

Part B
Armv8-M Architecture Rules

Chapter B1

Resets

This chapter specifies the Armv8-M reset rules. It contains the following section:

[B1.1 *Resets, Cold reset, and Warm reset* on page 47.](#)

B1.1 Resets, Cold reset, and Warm reset

- R_{BDDL}** There are two resets:
- Cold reset.
 - Warm reset.
- R_{CTPC}** It is not possible to have a Cold reset without also having a Warm reset.
- R_{FNNX}** On a Cold reset, registers that have a defined reset value contain that value.
- R_{GTXW}** On a Warm reset, some debug register control fields that have a defined reset value remain unchanged, but otherwise all registers that have a defined reset value contain that value.
- R_{YMHN}** On a Warm reset, the PE performs the actions that are described by the [TakeReset \(\)](#) pseudocode.
- R_{WSZN}** [AIRCR.SYSRESETREQ](#) is used to request a Warm reset.
- R_{HFRS}** For [AIRCR.SYSRESETREQ](#), the architecture does not guarantee that the reset takes place immediately.

See also:

[Chapter B11 Debug on page 230.](#)

Chapter B2

Power Management

This chapter specifies the Armv8-M power management rules. It contains the following section:

[B2.1 *Power management* on page 49.](#)

[B2.2 *Sleep on exit* on page 51.](#)

B2.1 Power management

I_{HCVL} The following instructions and pseudocode functions hint to the PE hardware that it can suspend execution and enter a low-power state:

- `WaitForEvent()`.
- `WaitForInterrupt()`.
- `SleepOnExit()`.

B2.1.1 The Wait for Event (WFE) instruction

R_{DCMH} When a [WFE](#) instruction is executed, if the state of the Event register is clear, the PE can suspend execution and enter a low-power state.

R_{HDXV} When a [WFE](#) instruction is executed, if the state of the Event register is set, the instruction clears the register and completes immediately.

R_{KDND} If the PE enters a low-power state on a WFE instruction, it remains in that low-power state until it receives a *WFE wakeup event*. When the PE recognizes a WFE wakeup event, the WFE instruction completes. The following are WFE wakeup events:

- The execution of a [SEV\(\)](#) instruction by any PE.
- When [SCR.SEVONPEND](#) is 1, any exception entering the pending state.
- Any exception at a priority that would preempt the current execution priority, taking into account any active exceptions and including the effects of any software-controlled priority boosting by [AIRCR.PRIS == 1](#) and [PRIMASK](#), [FAULTMASK](#), or [BASEPRI](#).
- If debug is enabled, a debug event.
- Any IMPLEMENTATION DEFINED event.

R_{YRDC} The Armv8-M architecture does not define the exact nature of the low-power state that is entered on a instruction, except that it does not cause a loss of [memory coherency](#).

I_{TZJZ} Arm recommends that software always uses the instruction in a loop.

See also:

[B3.13 Priority model on page 77](#).

[WaitForEvent\(\)](#).

[SendEvent\(\)](#).

B2.1.2 The Event register

I_{RPZM} The Event register is a single-bit register for each PE in the system.

- R_{BPBR}** The Event register for a PE is set by any of the following:
- Any WFE wakeup event.
 - Exception entry.
 - Exception return.
- I_{MMZW}** When the Event register is set, it is an indication that an event has occurred since the register was last cleared, and that the event might require some action by the PE.
- R_{CXMT}** A reset clears the Event register.
- I_{JFKL}** The execution of a [WFE](#) instruction will clear the Event Register.
- I_{LNFV}** Software cannot read, and cannot write to, the Event register directly.

See also:

[SetEventRegister\(\)](#)

[ClearEventRegister\(\)](#)

[EventRegistered\(\)](#)

B2.1.3 The Wait for Interrupt (WFI) instruction

- R_{HRMJ}** When a [WFI](#) instruction is executed, the PE can suspend execution and enter a low-power state. If it does, it remains in that state until it receives a *WFI wakeup event*. When the PE recognizes a [WFI](#) wakeup event, the [WFI](#) instruction completes. The following are [WFI](#) wakeup events:
- A reset.
 - Any asynchronous exception at a priority that, ignoring the effect of [PRIMASK](#) (so that behavior is as if [PRIMASK](#) is 0), would preempt any currently active exceptions.
 - An IMPLEMENTATION DEFINED [WFI](#) wakeup event.
 - If debug is enabled, a debug event.
- I_{CGNL}** Arm recommends that software always uses the [WFI](#) instruction in a loop.

See also:

[B3.13 Priority model on page 77.](#)

[WaitForInterrupt\(\)](#)

B2.2 Sleep on exit

- R_{JXGW}** It is IMPLEMENTATION DEFINED whether the [SleepOnExit \(\)](#) function causes the PE to enter a low-power state during the return from the only active exception and the PE returns to Thread mode.
- R_{CMVG}** The PE enters a low-power state on return from an exception when all the following are true:
- [EXC_RETURN.Mode](#) == 1.
 - [SCR.SLEEPONEXIT](#) == 1.
- R_{WWDW}** If the sleep-on-exit function is enabled, it is IMPLEMENTATION DEFINED at which point in the exception return process the PE enters a low-power state.
- R_{LLQF}** The wakeup events for the sleep-on-exit function are identical to the [WFI](#) instruction wakeup events.

See also:

[B3.13 Priority model on page 77.](#)

[SleepOnExit \(\)](#)

[B3.22 Exception return on page 101.](#)

Chapter B3

Programmers' Model

This chapter specifies the Armv8-M programmers' model architecture rules. It contains the following sections:

- [B3.1 PE modes, Thread mode and Handler mode on page 54.](#)
- [B3.2 Privileged and unprivileged execution on page 55.](#)
- [B3.3 Registers on page 56.](#)
- [B3.4 Special-purpose CONTROL register on page 58.](#)
- [B3.5 XPSR, APSR, IPSR, and EPSR on page 59.](#)
- [B3.6 Security states: Secure state, and Non-secure state on page 61.](#)
- [B3.7 Security states and register banking between Security states on page 62.](#)
- [B3.8 Stack pointer on page 63.](#)
- [B3.9 Exception numbers and exception priority numbers on page 65.](#)
- [B3.10 Exception enable, pending, and active bits on page 68.](#)
- [B3.11 Security states, exception banking on page 70.](#)
- [B3.12 Faults on page 72.](#)
- [B3.13 Priority model on page 77.](#)
- [B3.14 Secure address protection on page 81.](#)
- [B3.15 Security state transitions on page 82.](#)
- [B3.16 Function calls from Secure state to Non-secure state on page 84.](#)
- [B3.17 Function returns from Non-secure state on page 85.](#)

- B3.18 *Exception handling* on page 87.
- B3.19 *Exception entry, context stacking* on page 89.
- B3.20 *Exception entry, register clearing after context stacking* on page 97.
- B3.21 *Stack limit checks* on page 98.
- B3.22 *Exception return* on page 101.
- B3.23 *Integrity signature* on page 104.
- B3.24 *Exceptions during exception entry* on page 105.
- B3.25 *Exceptions during exception return* on page 107.
- B3.26 *Tail-chaining* on page 108.
- B3.27 *Exceptions, instruction resume, or instruction restart* on page 111.
- B3.28 *Vector tables* on page 114.
- B3.29 *Hardware-controlled priority escalation to HardFault* on page 116.
- B3.30 *Special-purpose mask registers, PRIMASK, BASEPRI, FAULTMASK, for configurable priority boosting* on page 117.
- B3.31 *Lockup* on page 119.
- B3.32 *Context Synchronization Event* on page 124.
- B3.33 *Coprocessor support* on page 125.

B3.1 PE modes, Thread mode and Handler mode

R_{CNMS} There are two PE modes:

- Thread mode.
- Handler mode.

I_{FDVT} A common usage model for the PE modes is:

- **Thread mode:** Applications.
- **Handler mode:** OS kernel and associated functions, that manage system resources.

R_{RPKP} The PE handles all exceptions in Handler mode.

R_{CMQP} Thread mode is selected on reset.

See also:

[B3.2 Privileged and unprivileged execution on page 55.](#)

[B3.5.1 Interrupt Program Status Register \(IPSR\) on page 59.](#)

[B3.6 Security states: Secure state, and Non-secure state on page 61.](#)

B3.2 Privileged and unprivileged execution

R_{WVRK}

Thread mode

Execution can be privileged or unprivileged.

Handler mode

Execution is always privileged.

I_{WCFH}

[CONTROL.nPRIV](#) determines whether execution in Thread mode is unprivileged.

R_{SBQF}

In a PE without the Main Extension, it is IMPLEMENTATION DEFINED whether [CONTROL.nPRIV](#) can be set to 1.

R_{JSSW}

Execution privilege can determine whether a resource is accessible.

I_{GNSC}

Privileged execution typically has access to more resources than unprivileged execution.

See also:

[B3.1 PE modes, Thread mode and Handler mode on page 54.](#)

B3.3 Registers

R_{KGST} There are the following types of registers:

General-purpose registers, all 32-bit:

- R0-R12 (**R_n**).
- R13. This is the stack pointer (**SP**).
- R14. This is the Link Register (**LR**).

Program Counter, 32-bit:

- R15 is the Program Counter (**PC**).

Special-purpose registers:

- Mask Registers:
 - 1-bit exception mask register, **PRIMASK**.
 - 8-bit base priority mask register, **BASEPRI**.
 - 1-bit fault mask register, **FAULTMASK**.
- A 2-bit, 3-bit, or 4-bit **CONTROL** register.
- Two 32-bit stack pointer limit registers, **MSPLIM** and **PSPLIM**, if the Main Extension is not implemented the Non-secure versions of these registers are RAZ/WI.
- A combined 32-bit Program Status Register (**XPSR**), comprising:
 - Application Program Status Register (**APSR**).
 - Interrupt Program Status Register (**IPSR**).
 - Execution Program Status Register (**EPSR**).

Memory-mapped registers:

All other registers.

I_{CJWV} A 32-bit combined exception return Program Status Register, **RETPSR**, contains a payload of the saved state derived from the **XPSR**.

I_{DHVL} Extensions might add more registers to the **Base register** set.

I_{BLXF} **SP** refers to the active stack pointer, the Main stack pointer or the Process stack pointer.

R_{PLRT} If the Main Extension is implemented, the **LR** is set to 0xFFFFFFFF on Warm reset.
*The extension requirements are - **M**.*

R_{QHMH} If the Main Extension is not implemented, the **LR** becomes UNKNOWN on a Warm reset.
*The extension requirements are - **!M**.*

R_{PLNS} The **PC** is loaded with the reset handler start address on Warm reset.

R_{JPCB} The **PC** contains the instruction address of the instruction currently being executed. If an instruction reads the

value of the **PC**, the value returned will increase by 4.

- R_{XHHC}** Except for writes to the **CONTROL** register, any change to a special-purpose register by a **CPS** or **MSR** instruction is guaranteed:
- Not to affect that **CPS** or **MSR** instruction, or any instruction preceding it in program order.
 - To be visible to all instructions that appear in program order after the **CPS** or **MSR**.
- R_{XPTQ}** All unallocated or reserved values of fields with allocated values within the memory-mapped registers that are described in this reference manual behave, unless otherwise stated in the register description, in one of the following ways:
- The encoding maps onto any of the allocated values, but otherwise does not cause **CONSTRAINED UNPREDICTABLE** behavior.
 - The encoding causes effects that could be achieved by a combination of more than one of the allocated encodings.
 - The encoding causes the field to have no functional effect.
- R_{PDJC}** Reads of registers described as write-only (**WO**) behave as **RES0**.
- R_{XKXP}** Writes to a register described as read-only (**RO**) do not cause modification of the read-only register.

See also:

[Chapter B6 The System Address Map on page 198.](#)

[B3.30 Special-purpose mask registers, *PRIMASK*, *BASEPRI*, *FAULTMASK*, for configurable priority boosting on page 117.](#)

[B3.4 Special-purpose *CONTROL* register on page 58.](#)

[B3.21 Stack limit checks on page 98.](#)

[B3.5 *XPSR*, *APSR*, *IPSR*, and *EPSR* on page 59.](#)

[B1.1 Resets, Cold reset, and Warm reset on page 47.](#)

[Chapter D1 Register Specification.](#)

B3.4 Special-purpose CONTROL register

- R_{CSPP}** [MRS](#) and [MSR](#) instructions can be used to access the [CONTROL](#) register.
- R_{GKVQ}** Privileged execution can write to the [CONTROL](#) register. The PE ignores unprivileged writes to the [CONTROL](#) register. All reads of the [CONTROL](#) register, regardless of privilege, are allowed.
- R_{RJMP}** The architecture requires a [Context synchronization event](#) to guarantee visibility of a change to the [CONTROL](#) register.
- R_{HVGB}** The PE automatically updates [CONTROL.SPSEL](#) on exception entry and exception return.
- I_{NMBL}** [CONTROL.SPSEL](#) selects the stack pointer when the PE is in Thread mode.

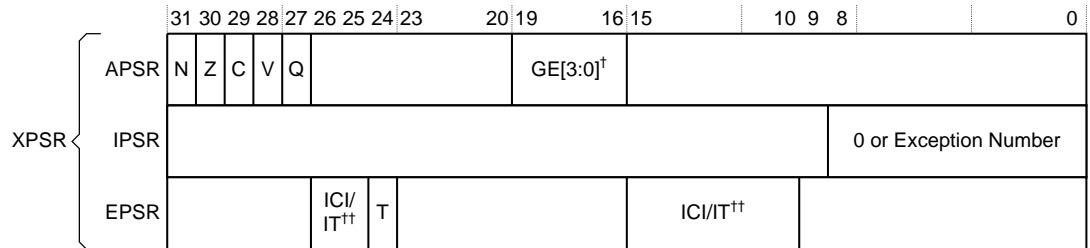
See also:

[B3.32 Context Synchronization Event on page 124.](#)

[CONTROL, Control Register.](#)

B3.5 XPSR, APSR, IPSR, and EPSR

R_{VWTF} The **APSR**, **IPSR** and **EPSR** combine to form one register, the **XPSR**:



† Reserved if the DSP Extension is not implemented
 †† Reserved if the Main Extension is not implemented

All unused bits in any of the **APSR**, **IPSR**, or **EPSR**, or any unused bits in the combined **XPSR**, are reserved.

R_{XGTF} The **MRS** and **MSR** instructions recognize the following mnemonics for accessing the **APSR**, **IPSR** or **EPSR**, or a combination of them:

Mnemonic	Registers accessed
APSR	APSR
IPSR	IPSR
EPSR	EPSR
IAPSR	IPSR and APSR
EAPSR	EPSR and APSR
IEPSR	IPSR and EPSR
XPSR	APSR, IPSR, and EPSR

R_{WLF} Arm deprecates using **MSR APSR** without a `<bits>` qualifier as an alias for **MSR APSR-`_nzcvcq`**.

See also:

[B3.3 Registers on page 56.](#)

[APSR, Application Program Status Register.](#)

[B3.5.1 Interrupt Program Status Register \(IPSR\) .](#)

[B3.5.2 Execution Program Status Register \(EPSR\) on page 60.](#)

B3.5.1 Interrupt Program Status Register (IPSR)

R_{DTBJ} When the PE is in Thread mode, the **IPSR** value is zero.

When the PE is in Handler mode:

- In the case of a taken exception, the **IPSR** holds the exception number of the exception being handled.
- When there has been a function call from Secure state to Non-secure state, the **IPSR** has the value of 1.

The PE updates the **IPSR** on exception entry and return.

Note, Secure state requires S.

R_{XTCC} The PE ignores writes to the **IPSR** by **MSR** instructions.

R_{RDRX} If **DCRDR** is used to change the value of **IPSR**, then the value of **IPSR** becomes UNKNOWN. If **DCRDR** attempts to set **IPSR** to an illegal value, then the UNKNOWN value is set to one of the known legal values.

See also:

[B3.5 XPSR, APSR, IPSR, and EPSR on page 59.](#)

[B3.16 Function calls from Secure state to Non-secure state on page 84.](#)

[IPSR, Interrupt Program Status Register](#)

[BX](#), [BXNS](#)

B3.5.2 Execution Program Status Register (EPSR)

R_{KSCH} A reset sets **EPSR.T** to the value of bit[0] of the reset vector.

R_{SQIX} When **EPSR.T** is:

0: Any attempt to execute any instruction generates:

- An **INVSTATE UsageFault**, in a PE with the Main Extension.
- A **HardFault**, in a PE without the Main Extension.

1: The Instruction set state is T32 state and all instructions are decoded as T32 instructions.

Note, UsageFault requires M.

I_{XBWX} The intent is that the Instruction set state is always T32 state.

R_{LBJQ} All **EPSR** fields read as zero using an **MRS** instruction. The PE ignores writes to the **EPSR** by an **MSR** instruction.

See also:

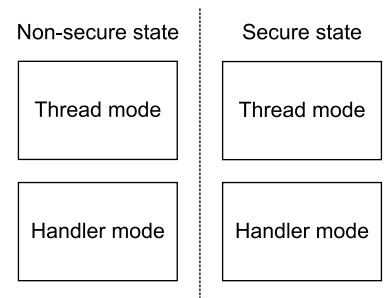
[B3.5 XPSR, APSR, IPSR, and EPSR on page 59.](#)

[B3.5.2 Execution Program Status Register \(EPSR\).](#)

B3.6 Security states: Secure state, and Non-secure state

R_{HKKL} A PE with the Security Extension has two Security states:

- Secure state.
 - Secure Thread mode.
 - Secure Handler mode.
- Non-secure state.
 - Non-secure Thread mode.
 - Non-secure Handler mode.



The extension requirements are - S.

R_{PBGT} If the Security Extension is implemented, memory areas and other critical resources that are marked as secure can only be accessed when the PE is executing in Secure state.

The extension requirements are - S.

R_{HWFV} A PE with the Security Extension resets into Secure state on both of the Armv8-M resets, Cold reset and Warm reset.

The extension requirements are - S.

R_{PLGH} A PE without the Security Extension resets into Non-secure state on both of the Armv8-M resets, Cold reset and Warm reset.

The extension requirements are - !S.

See also:

[B3.1 PE modes, Thread mode and Handler mode on page 54.](#)

[B3.2 Privileged and unprivileged execution on page 55.](#)

[B3.7 Security states and register banking between Security states on page 62.](#)

[B3.11 Security states, exception banking on page 70.](#)

[B3.15 Security state transitions on page 82.](#)

B3.7 Security states and register banking between Security states

I_{MGRQ} In a PE with the Security Extension, some registers are banked between the Security states. When a register is banked in this way, there is a distinct instance of the register in Secure state and another distinct instance of the register in Non-secure state.

The extension requirements are - [S](#).

R_{BHDK} In a PE with the Security Extension:

- The general-purpose registers that are banked are:
 - R13. This is the stack pointer ([SP](#)).
- The special-purpose registers that are banked are:
 - The Mask registers, PRIMASK, BASEPRI, and FAULTMASK.
 - Some bits in the CONTROL register.
 - The Main and Process stack pointer Limit registers, [MSPLIM](#) and [PSPLIM](#).
- The [System Control Space](#) (SCS) is banked.

The extension requirements are - [S](#).

I_{GBWT} For [MRS](#) and [MSR \(register\)](#) instructions, SYSm[7] in the instruction encoding specifies whether the Secure or the Non-secure instance of a [Banked register](#) is accessed:

Access from	SYSm[7]	
	0	1
Secure state	Secure instance	Non-secure instance
Non-secure state	Non-secure instance	RAZ/WI

The extension requirements are - [S](#).

R_{ZJKL} This specification uses the following naming convention to identify a [Banked register](#):

- **<register name>_S**: The Secure instance of the register.
- **<register name>_NS**: The Non-secure instance of the register.
- **<register name>**: The instance that is associated with the current Security state.

The extension requirements are - [S](#).

See also:

[B3.3 Registers](#) on page 56.

[B3.6 Security states: Secure state, and Non-secure state](#) on page 61.

[B3.8 Stack pointer](#) on page 63.

[B6.3 The System Control Space \(SCS\)](#) on page 202.

B3.8 Stack pointer

R_{RDLR} In a PE with the Security Extension, four stacks and four stack pointer registers are implemented:

Stack	Stack pointer register	
Secure	Main	MSP_S
	Process	PSP_S
Non-secure	Main	MSP_NS
	Process	PSP_NS

The extension requirements are - **S**.

R_{TGHV} In a PE without the Security Extension, two stacks and two stack pointer registers are implemented:

Stack	Stack pointer register
Main	MPS
Process	PSP

The extension requirements are - **!S**.

R_{LDGJ} On exception return the Armv8-M architecture only supports doubleword aligned stack pointers.

R_{XKZV} If, on exception return, the stack pointers are not doubleword aligned, the CONstrained UNpredictable behavior is either:

- Treating the stack pointer as the actual value.
- Treating the stack pointer as if it were aligned.

R_{TXRW} In Handler mode, the PE uses the main stack.

I_{DMLS} In Thread mode, **CONTROL.SPSEL** determines whether the PE uses the main or process stack.

R_{BTVD} In a PE without the Security Extension, MSP is selected and initialized on reset.

The extension requirements are - **!S**.

R_{MDXK} In a PE with the Security Extension, the Secure main stack, MSP_S, is selected and initialized on reset.

The extension requirements are - **S**.

I_{LVWN} On Warm reset, the selected Stack Pointer either the MSP or MSP_S, is set to the value contained in the Vector table, as described in **TakeReset ()**.

Note, S is required for MSP_S.

R_{XPWM} Bits [1:0] of the MSP or PSP, in either Security state, are **RES0H**, so that all stack pointers are always guaranteed to be word-aligned.

The extension requirements are - **S**.

R_{MQVJ} Where an instruction states that the **SP** is UNpredictable and **SP** is used:

- The value that is read or written from or to the **SP** is UNKNOWN.
- The instruction is permitted to be treated as UNDEFINED.

- If the [SP](#) is being written, it is UNKNOWN whether a stack-limit check is applied.

R_{JXJM}

After the successful completion of an exception entry stacking operation, the stack pointer of the stack pushed because of the exception entry is doubleword-aligned.

I_{PWRQ}

Arm recommends that the Secure stacks be located in Secure memory.

The extension requirements are - [S](#).

See also:

[B3.6 Security states: Secure state, and Non-secure state on page 61.](#)

[B3.1 PE modes, Thread mode and Handler mode on page 54.](#)

[B3.19 Exception entry, context stacking on page 89.](#)

[B3.28 Vector tables on page 114.](#)

[B3.3 Registers on page 56.](#)

[B3.21 Stack limit checks on page 98.](#)

B3.9 Exception numbers and exception priority numbers

I_{DCJS} Each exception has an associated *exception number* and an associated *priority number*.

R_{CMTC} In a PE with the Main Extension, the exceptions, their associated numbers, and their associated priority numbers are as follows:

Exception	Exception Number	Priority Number
Reset	1	-4 (Highest Priority)
Secure HardFault when AIRCR.BFHFNMINs is 1	3	-3
NMI	2	-2
Secure HardFault when AIRCR.BFHFNMINs is 0	3	-1
Non-Secure HardFault	3	-1
MemManage fault	4	Configurable
BusFault	5	Configurable
UsageFault	6	Configurable
SecureFault	7	Configurable
Reserved	8-10	-
SVCall	11	Configurable
DebugMonitor	12	Configurable
Reserved	13	-
PendSV	14	Configurable
SysTick	15	Configurable
External Interrupt 0	16	Configurable
-	-	-
-	-	-
-	-	-
External interrupt N	16+N	Configurable

When [AIRCR.BFHFNMINs](#) is 1, faults that target Secure state that are escalated to HardFault are still Secure HardFaults. That is, the value of [AIRCR.BFHFNMINs](#) does not affect faults that target Secure state that are escalated to HardFaults. This table row applies to such faults.

If the Security Extension is not implemented exception 7 is reserved.

The extension requirements are - M. Note, S is required for Secure faults.

R_{MGNV} In a PE without the Main Extension, the exceptions, their associated numbers, and their associated priority numbers are as follows:

Exception	Exception Number	Priority Number
Reset	1	-4 (Highest Priority)
Secure HardFault when AIRCR.BFHFNMINs is 1	3	-3
NMI	2	-2
Secure HardFault when AIRCR.BFHFNMINs is 0	3	-1
Non-Secure HardFault	3	-1
Reserved	4-10	-
SVCall	11	Configurable
Reserved	12-13	-
PendSV	14	Configurable
SysTick	15	Configurable
External Interrupt 0	16	Configurable
-	-	-
-	-	-
-	-	-
External interrupt N	16+N	Configurable

When [AIRCR.BFHFNMINs](#) is 1, faults that target Secure state that are escalated to HardFault are still Secure HardFaults. That is, the value of [AIRCR.BFHFNMINs](#) does not affect faults that target Secure state that are escalated to HardFaults. This table row applies to such faults.

The extension requirements are - *!M*. Note, *S* is required for Secure faults. *ST* is required for SysTick fault.

I_{FPJD} The maximum supported number of external interrupts is 496, regardless of whether the Main Extension is implemented.

R_{QOTT} The architecture permits an implementation to omit external configurable interrupts where no external device is connected to the corresponding interrupt pin. Where an implementation omits such an interrupt, the corresponding pending, active, enable, and priority registers are RES0.

I_{QWTM} In a PE with the Main Extension, the following exceptions with configurable priority numbers can be configured with [SHPR1](#)- [SHPR3](#) in the System Control Block (SCB):

- MemManage Fault.
- BusFault.
- UsageFault.
- SecureFault (if the Security Extension is implemented).
- SVCall.
- DebugMonitor exception.
- PendSV.
- SysTick.

The extension requirements are - *M*.

I_{SGBC} In a PE without the Main Extension the following exceptions with configurable priority numbers can be configured with [SHPR2](#) and [SHPR3](#) in the System Control Block (SCB):

- SVCall.
- PendSV.
- SysTick.

The extension requirements are - *!M*.

I_{JQPH} All other configurable exceptions can be configured using the [NVIC_IPRn.PRI_<n>](#) register fields.

R_{NFSM} Configurable priority numbers start at 0, the highest configurable exception priority number.

R_{GGCP} In a PE with the Main Extension, the number of configurable priority numbers is an IMPLEMENTATION DEFINED power of two in the range 8-256:

Number of priority bits of SHPRIn.PRI_n implemented	Number of configurable Priority numbers	Minimum Priority Number (highest priority)	Maximum Priority Number (lowest priority)
3	8	0	0b111100000 = 224
4	16	0	0b111110000 = 240
5	32	0	0b111111000 = 248
6	64	0	0b111111100 = 252
7	128	0	0b111111110 = 254
8	256	0	0b111111111 = 255

All low-order bits of SHPRIn.PRI_n that are not implemented as priority bits are RES0, as shown in the maximum priority number column.

The extension requirements are - *M*.

R_{CMGH} In a PE without the Main Extension, the number of configurable priority numbers is 4:

Number of priority bits of SHPRIn.PRI_n implemented	Number of configurable Priority numbers	Minimum Priority Number (highest priority)	Maximum Priority Number (lowest priority)
2	4	0	0b11000000 = 192

SHPRn.PRI_n[5:0] are RES0, as shown in the maximum priority number column.

The extension requirements are - *!M*.

See also:

[B3.11 Security states, exception banking on page 70.](#)

[B3.12 Faults on page 72.](#)

[B3.13 Priority model on page 77.](#)

[SHPR1, SHPR2, SHPR3.](#)

[NVIC_IPRn.](#)

[ExecutionPriority\(\)](#)

B3.10 Exception enable, pending, and active bits

I_{QODG} The [SHCSR](#), [ICSR](#), [DEMCR](#), [NVIC_IABRn](#), [NVIC_ISPRn](#) contain exception enable, pending, and active fields. [STIR](#) can be used to pend exceptions.

I_{GHW} The following exceptions are always enabled and therefore do not have an exception enable bit:

- HardFault.
- NMI.
- SVCcall.
- PendSV.

I_{LHSX} In a PE without the Security Extension:

- Privileged execution can pend interrupts by writing to the [NVIC_ISPRn](#).
- When [CCR.USERSETMPEND](#) is 1, unprivileged execution can pend interrupts by writing to the [STIR](#).

The extension requirements are - !S.

I_{QDKX} In a PE with the Security Extension:

- The [STIR](#) can pend any Secure or Non-secure interrupt, as follows:

	Secure state	Non-secure state
Privileged execution	Can use STIR to pend any Secure or Non-secure interrupt.	Can use STIR to pend a Non-Secure interrupt.
Unprivileged execution	When CCR_S.USERSETMPEND is 1, can use STIR to pend any Secure or Non-secure interrupt, otherwise when CCR_S.USERSETMPEND is 0 a BusFault is generated.	When CCR_NS.USERSETMPEND is 1 can use STIR to pend any Non-secure interrupt, otherwise when CCR_NS.USERSETMPEND is 0 a BusFault is generated.

- The [STIR_NS](#) can pend a Non-secure interrupt, as follows:

	Secure state	Non-secure state
Privileged	Can use STIR_NS to pend a Non-secure interrupt.	RES0
Unprivileged	When CCR_NS.USERSETMPEND is 1, can use STIR_NS to pend a Non-secure interrupt, otherwise when CCR_NS.USERSETMPEND is 0 a BusFault is generated.	BusFault

- The [NVIC_ISPRn](#) can pend any Secure or Non-secure interrupt, as follows:

	Secure state	Non-secure state
Privileged execution	Can use NVIC_ISPRn to pend any Secure or Non-secure interrupt	Can use NVIC_ISPRn to pend a Non-secure interrupt
Unprivileged execution	BusFault	BusFault

- The [NVIC_ISPRn_NS](#) can pend a Non-secure interrupt, as follows:

	Secure state	Non-secure state
Privileged execution	Can use <code>NVIC_ISPRn_NS</code> to pend a Non-secure interrupt	RES0
Unprivileged execution	BusFault	BusFault

The extension requirements are - [S](#).

I_TRJJ

The following table identifies the fault enable, status and active bits:

Fault, Enable (SHCSR) and Trap Bits	Status bit	Pending bit SHCSR, ICSR	Active bit SHCSR
Secure HardFault	HFSR.VECTTBL HFSR.FORCED HFSR.DEBUGEVT	HARDFFAULTPENDED	HARDFFAULTACT
NMI	-	PENDNMISET	NMIACT
HardFault	HFSR.VECTTBL HFSR.FORCED HFSR.DEBUGEVT	HARDFFAULTPENDED	HARDFFAULTACT
MemmanageFault MEMFAULTENA	MMFSR.IACCVIOL MMFSR.DACCVIOL MMFSR.MUNSTKERR MMFSR.MSTKERR MMFSR.MLSPERR	MEMFAULTPENDED	MEMFAULTACT
BusFault BUSFAULTENA	BFSR.IBUSERR BFSR.PRECISERR BFSR.IMPRESERR BFSR.UNSTKERR BFSR.STKERR BFSR.LSPERR	BUSFAULTPENDED	BUSFAULTACT
UsageFault	UFSR.UNDEFINSTR UFSR.INVSTATE UFSR.INVPC UFSR.NOCP UFSR.STKOF	USGFAULTPENDED	USGFAULTACT
CCR.UNALIGN_TRP	UFSR.UNALIGNED	-	-
CCR.DIV_0_TRP	UFSR.DIVBYZERO	-	-
SecureFault SECUREFAULTENA	SFSR.INVEP SFSR.INVIS SFSR.INVER SFSR.AUVIOL SFSR.INVTRAN SFSR.LSPERR SFSR.LSERR	SECUREFAULTPENDED	SECUREFAULTACT
SVCall	-	SVCALLPENDED	SVCALLACT
DebugMonitor DEMCR.MON_EN	-	DEMCR.MON_PEND	MONITORACT
PendSV	-	PENDSVSET	PENDSVACT
SysTick SYST_CSR.ENABLE and SYST_CSR.TICKINT	-	PENDSTSET	SYSTICKACT
External Interrupt NVIC_ICERn	-	NVIC_ISPRn NVIC_ICPRn	NVIC_IABRn

B3.11 Security states, exception banking

R_{PJHV} Some exceptions are banked. A banked exception has all the following:

- Banked enabled, pending, and active bits.
- A banked **SHPR_n.PRI** field.
- A banked exception vector.
- A state relevant handler.

Exception	Banked
Reset	No
HardFault	Yes (conditionally)
NMI	No
MemManage fault	Yes
BusFault	No
UsageFault	Yes
SecureFault	No
SVCall	Yes
DebugMonitor	No
PendSV	Yes
SysTick	Yes
External interrupt 0	No
-	-
-	-
-	-
External interrupt N	No

MemManage Fault, UsageFault, BusFault and the DebugMonitor exception require the Main Extension to be implemented. SecureFault requires the Security Extension to be implemented. The SysTick exception is banked if the Main Extension is implemented. If the Main Extension is not implemented, it is IMPLEMENTATION DEFINED if the exception is banked or if there is a single instance that has a configurable target Security state.

Note, some exceptions require M, S, DebugMonitor exception or ST.

R_{LNWV} A banked synchronous exception targets the Security state that it is taken from, except for the following cases:

- When accessing a coprocessor that is disabled only by the **NSACR**, any NOCP UsageFault that is generated as a result of that access will target Secure state, even though the PE was executing in Non-secure state.
- When accessing a coprocessor that is disabled by the **CPPWR**, any NOCP UsageFault that is generated as a result of that access will target the Secure state if the corresponding **CPPWR.SUS_m** bit is set to 1, otherwise the NOCP UsageFault will target the current Security state.
- If an instruction triggers lazy floating-point state preservation, then the banked exception will be raised as if the current Security state was the same as that of the floating-point state, as indicated by **FPCCR.S**.
- Banked faults and exceptions which arise from instruction fetch will target the Security state associated with the instruction address instead of the current Security state.
- Where Non-secure HardFault is enabled, because **AIRCR.BFHFNMINS** is set to 1, the following applies:
 - HardFault exceptions generated through escalation will target the Security state of the original exception before its escalation to HardFault.
 - A HardFault generated as a result of a failed vector fetch will target the Security state of the exception raised during the failed vector fetch and not the current Security state.
- Faults triggered by the stacking of callee registers always target the Secure state.

The extension requirements are - S. Note, a UsageFault requires M, Floating-point state requires FP.

- R_{GVPG}** If [AIRCR.BFHFNMINs](#) == 0, then all Non-secure HardFaults are escalated to Secure HardFaults, and Non-secure pending bits behave as zero for everything except explicit reads and writes.
- R_{WLGH}** Where an implementation has two SysTick timers, one in each Security state, each timer targets its owning Security state and not the current Execution state of the PE.
The extension requirements are - [S](#) && [ST](#).
- I_{DDKC}** NMI can be configured to target either Security state, by using [AIRCR.BFHFNMINs](#).
- I_{HGFM}** BusFault can be configured to target either Security state, by using [AIRCR.BFHFNMINs](#).
- R_{MQWN}** SecureFault always targets Secure state.
The extension requirements are - [S](#).
- I_{WSSL}** The DebugMonitor exception targets Secure state if the status bit [DEMCR.SDME](#) is 1. Otherwise, it targets Non-secure state.
- I_{DQLX}** Each external interrupt, 0-N, targets the Security state that its [NVIC_ITNSn](#)[bit number] dictates.
- R_{HXRW}** When <exception> targets Secure state, the Non-secure view of its priority field, and enabled, pending, and active bits, are RAZ/WI.
<exception> is one of:
- NMI.
 - BusFault.
 - DebugMonitor.
 - External interrupt N.
 - In a PE without the Main Extension, and a single instance of the SysTick Timer, SysTick.
- The extension requirements are - [S](#). Note, a BusFault exception requires [M](#), a DebugMonitor exception requires DebugMonitor exception.*
- I_{LFHQ}** Secure software must ensure that when changing the target Security state of an exception, the exception is not pending or active.

See also:

[B3.9 Exception numbers and exception priority numbers on page 65.](#)

[B3.28 Vector tables on page 114.](#)

[SHCSR, System Handler Control and State Register.](#)

B3.12 Faults

I_{NHTE}

There are the following Fault Status Registers:

- HardFault Status Register **HFSR**. Present only if the Main Extension is implemented.
- MemManage Fault Status Register **MMFSR**. Present only if the Main Extension is implemented.
- BusFault Status Register **BFSR**. Present only if the Main Extension is implemented.
- UsageFault Status Register **UFSR**. Present only if the Main Extension is implemented.
- SecureFault Status Register **SFSR**. Present only if the Main Extension is implemented.
- Debug Fault Status Register **DFSR**. Present only if Halting debug or the Main Extension is implemented.
- Auxiliary Fault Status Register **AFSR**. The contents of this register are IMPLEMENTATION DEFINED.

In a PE with the Main Extension, the **BFSR**, **MMFSR**, and **UFSR** combine to form one register, called the Configurable Fault Status Register (**CFSR**).

There are the following Fault Address Registers:

- MemManage Fault Address Register (**MMFAR**). Present only if the Main Extension is implemented.
- BusFault Address Register (**BFAR**). Present only if the Main Extension is implemented.
- SecureFault Address Register (**SFAR**). Present only if the Main Extension is implemented.

The extension requirements are - **M**.

R_{XMRH}

Unless otherwise stated, **MMFAR** is updated only for a MemManage fault on a direct data access.

The extension requirements are - **M**.

R_{DDJJ}

Unless otherwise stated, **BFAR** is updated only for a BusFault on a data access, a precise fault.

The extension requirements are - **M**.

R_{BFFR}

Unless otherwise stated, **SFAR** is updated only for a SecureFault on a memory access that caused a Security Attribution Unit violation.

The extension requirements are - **M && S**.

R_{FLDT}

Each fault address register has an associated valid bit. When the PE updates the fault address register, the PE sets the valid bit to 1.

Fault address register	Valid bit
MMFAR	MMFSR.MMARVALID
BFAR	BFSR.BFARVALID
SFAR	SFSR.SFARVALID

The extension requirements are - **M**.

R_{TSCP}

If a fault occurs that would otherwise set a FAR valid bit to 1, which has already been set by an earlier fault it is IMPLEMENTATION DEFINED whether:

- The FAR register is updated with the new syndrome.
- The FAR register retains the syndrome of the original fault.

R_{TSCG}

If the Security Extension is not implemented, it is IMPLEMENTATION DEFINED whether separate **BFAR** and **MMFAR** are implemented. If one shared fault address register is implemented, then on a fault that would otherwise update the shared fault address register, if one of the other valid bits is set to 1, it is IMPLEMENTATION DEFINED whether:

- The shared fault address register is updated, the valid bit for the fault is set, and the other valid bit is cleared.
- The shared fault address register is not updated, and the valid bits are not changed.

The extension requirements are - **M** && **!S**.

R_{XPWN}

If the Security Extension is implemented, it is IMPLEMENTATION DEFINED whether separate fault address registers for each fault are implemented. If one shared fault address register per Security state is implemented, then on a fault that would otherwise update the shared fault address register, if one of the other FAR valid bits for the same Security state is set to 1, it is IMPLEMENTATION DEFINED whether:

- The shared fault address register is updated, the valid bit for the fault is set, and the other valid bit is cleared.
- The shared fault address register is not updated, and the valid bits are not changed.

The extension requirements are - **M** && **S**.

I_{LQFZ}

AIRCR.BFHFNMINs dictates which shared fault registers are used depending on the Security state:

- In Secure state, the fault registers that might be shared are **SFAR** and **MMFAR_S**, and when **AIRCR.BFHFNMINs** is 0, **BFAR**.
- In Non-secure state, the fault registers that might be shared are **MMFAR_NS** and when **AIRCR.BFHFNMINs** is 1, **BFAR**.

The extension requirements are - **M** && **S**.

I_{DGDG}

For example in an implementation that contains a shared Fault Address Register, the shared fault register might be written to with the address of the memory location being accessed that caused a BusFault fault, and **BFSR.BFARVALID** is set to 1. If the **BFSR.BFARVALID** is still set when a MemManage fault subsequently occurs that might update the shared fault address register it is an IMPLEMENTATION DEFINED choice between:

- The shared fault address register being updated with the syndrome that caused the MemManage fault, the **BFSR.BFARVALID** being cleared and **MMFSR.MMARVALID** for the MemManage fault being set.
- The address of the memory location that caused the BusFault is retained and the fault address information generated by the MemManage fault is discarded without clearing **BFSR.BFARVALID** or updating **MMFSR.MMARVALID** for the MemManage fault.

The extension requirements are - **M** && **!S**.

I_{NSSB}

If a Secure MemManage fault occurs that targets Secure state, as described by **ExceptionTargetsSecure()**, the Secure shared fault register is written to with the address of the memory location that caused the MemManage fault, and **MMFSR_S.MMARVALID** is set to 1. If the **MMFSR_S.MMARVALID** is still set when a SecureFault subsequently occurs it is an IMPLEMENTATION DEFINED choice between:

- The shared fault address register being updated with the address of the memory location that that caused the SecureFault, **MMFSR_S.MMARVALID** being cleared and **SFSR.SFARVALID** being set.
- The address of the memory location that caused the MemManageFault is retained and the fault address information generated by the SecureFault is discarded without out clearing **MMFSR_S.MMARVALID** or updating **SFSR.SFARVALID**.

The extension requirements are - **M** && **S**.

I_{SMVQ}

Arm strongly recommends that if a separate **BFAR** is implemented, the associated **BFAR** and **BFSR.BFARVALID** bit is cleared when changing **AIRCR.BFHFNMINs** so as not to expose the last accessed address to Non-secure state.

The extension requirements are - **M** && **S**.

I_{LRNV}

When the Security extension is implemented:

- If **AIRCR.BFHFNMINs** is 0 a read of **BFAR_NS** will return 0.
- If **AIRCR.BFHFNMINs** is 1 a read of **BFAR_NS** and **BFAR_S** might return the same value. This behavior cannot be relied on if there is a change to **AIRCR.BFHFNMINs** between the reads of **BFAR** in different Security states.

The extension requirements are - **M** && **S**.

R_{KJPM} In a PE with the Main Extension, the faults are:

Exception Number	Exception		Fault Status Bit
3	HardFault	HardFault on Vector table entry read error	HFSR.VECTTBL
		HardFault on fault escalation	HFSR.FORCED
		HardFault on BKPT escalation	HFSR.DEBUGEVT
4	MemManage Fault	MemManage fault on an instruction fetch	MMFSR.IACCVIOL
		MemManage Fault on direct data access	MMFSR.DACCVIOL
		MemManage Fault on context unstacking by hardware.	MMFSR.MUNSTKERR
		MemManage Fault on context stacking by hardware, because of a MPU access violation.	MMFSR.MSTKERR
		When lazy Floating-point context preservation is active, a MemManage fault on saving Floating-point context to the stack	MMFSR.MLSPERR
5	BusFault	BusFault on an instruction fetch, precise	BFSR.IBUSERR
		BusFault on a data access, precise	BFSR.PRECISERR
		BusFault on a data access, imprecise	BFSR.IMPRESISERR
		BusFault on a context unstacking by hardware	BFSR.UNSTKERR
		BusFault on context stacking by hardware	BFSR.STKERR
		When lazy Floating-point context preservation is active, a BusFault on saving Floating-point context to the stack	BFSR.LSPERR
6	UsageFault	UsageFault, undefined instruction	UFSR.UNDEFINSTR
		UsageFault, invalid Instruction set state because EPSR.T is 0 or because of an exception return with a valid ICI value where the return address does not target either a load/store/clear multiple instruction or a breakpoint instruction	UFSR.INVSTATE
		UsageFault, failed integrity check on exception return or a function return with a transition from Non-secure state to Secure state	UFSR.INVPC
		UsageFault, no coprocessor	UFSR.NOCP
		UsageFault, stack overflow	UFSR.STKOF
		UsageFault, unaligned access	UFSR.UNALIGNED
		UsageFault, divide by zero when CCR.DIV_0_TRP is 1	UFSR.DIVBYZERO
7	SecureFault	SecureFault, invalid Secure state entry point	SFSR.INVEP
		SecureFault, invalid integrity signature when unstacking	SFSR.INVIS
		SecureFault, invalid exception return	SFSR.INVER
		SecureFault, attribution unit violation	SFSR.AUVIOL
		SecureFault, invalid transition from	SFSR.INVTRAN

Secure state	
SecureFault, lazy Floating-point context preservation error	SFSR.LSPERR
SecureFault, lazy Floating-point context error	SFSR.LSERR

The extension requirements are - **M**. Note, Secure Faults require **S**.

R_{XVNN} Exception vector reads use the default system address map.

I_{NKHG} In a PE without the Main Extension, the enable, pending, and active bits in [SHCSR](#) are RES0 for those faults that are only included in a PE with the Main Extension.

The extension requirements are - **M**.

R_{WHBK} In a PE without the Main Extension, the faults are:

Exception number	Exception
3	HardFault

The extension requirements are - **!M**.

R_{FQJV} Fault conditions that would generate a SecureFault in a PE with the Main Extension instead generate a Secure HardFault in a PE without the Main Extension.

The extension requirements are - **S**.

I_{CCXG} For the exact circumstances under which each of the Armv8-M faults is generated, see the appropriate Fault Status Register description.

The extension requirements are - **M**.

See also:

[B3.9 Exception numbers and exception priority numbers on page 65.](#)

[B3.29 Hardware-controlled priority escalation to HardFault on page 116.](#)

[Chapter B11 Debug on page 230.](#)

[Chapter D1 Register Specification.](#)

B3.13 Priority model

- I_{CTFJ}** An exception, other than reset, has the following possible states:
- Active:**
An exception that either:
- Is being handled.
 - Was being handled. The handler was preempted by a handler for a higher priority exception.
- Pending:**
An exception that has been generated, but that is not active.
- Inactive:**
The exception has not been generated.
- Active and pending:**
One instance of the exception is active, and a second instance of the exception is pending. Only asynchronous exceptions can be active and pending. Synchronous exceptions are either inactive, pending, or active.
- R_{CJDM}** Lower priority numbers take precedence over higher priority numbers.
- R_{HLJC}** When no exception is active and no priority boosting is active, the instruction stream that is executing has a priority number of (maximum supported priority number+1). The instruction stream that is executing can be interrupted by an exception with sufficient priority.
- If any exceptions are active the current execution priority is determined by:
1. In a PE with the Main Extension, the calculation of the effect of [AIRCR.PRIGROUP](#) on the comparison of [BASEPRI](#) to the [SHPRn.PRI](#) and [NVIC_IPRn](#) values.
 2. In a PE with or without the Main Extension applying the effects of [PRIMASK.PM](#) and [AIRCR.PRIS](#).
 3. In a PE with the Main Extension applying the effects of [FAULTMASK.FM](#).
 4. The execution priority is the either:
 - The exception with the lowest priority number.
 - The exception with the lowest priority group value.
- R_{RKCQ}** Execution at a particular priority can only be preempted by an exception with a lower group priority value.
- R_{LLCX}** In thread mode, when there are no active exceptions and no priority boosting is enabled, the execution priority is 256.
- I_{DPSP}** In a PE with the Main Extension, [BASEPRI](#) and each [SHPRn.PRI_n](#) and [NVIC_IPRn.PRI_Nn](#) are 8-bit fields that [AIRCR.PRIGROUP](#) splits into two fields, a group priority field and a subpriority field:

AIRCR.PRIGROUP value	BASEPRI, SHPRn.PRI_n [7:0], and NVIC_IPRn.PRI_Nn [7:0]	
	Group priority field	Subpriority field
0	[7:1]	[0]
1	[7:2]	[1:0]
2	[7:3]	[2:0]
3	[7:4]	[3:0]
4	[7:5]	[4:0]
5	[7:6]	[5:0]
6	[7]	[6:0]
7	-	[7:0]

In a PE without the Main Extension, **AIRCR.PRIGROUP** is RES0, therefore each SHPRn.PRI_n and NVIC_IPRn.PRI_Nn is split into two as follows:

AIRCR.PRIGROUP	SHPRn.PRI_n [7:0], and NVIC_IPRn.PRI_Nn [7:0]	
	Group priority field	Subpriority field
RES0	[7:1]	[0]

SHPRn.PRI_n[5:0] are RES0 in a PE without the Main Extension.

All low order bits of BASEPRI, **SHPRn.PRI**, and **NVIC_IPRn** are not implemented as priority bits are RES0.

R_{WORK}

When **AIRCR.PRIS** is 1, each Non-secure SHPRn_NS.PRI_n priority field value [7:0] has the following sequence applied to it, it:

1. Is divided by two.
2. The constant 0x80 is then added to it.

This is equivalent to the priority field value `output_pri = '1':input_pri[7:1]` and the priority comparisons are done on the effective field value after the division by 2 + 0x80 has been performed.

This maps the Non-secure SHPRn_NS.PRI_n group priority field values to the bottom half of the priority range. When this sequence is applied, any effects of **AIRCR.PRIGROUP** have already been taken into account, so the subpriority field is dropped and the sequence is only applied to the group priority field.

The extension requirements are - S. Note, Subpriority requires M.

R_{QORV}

After applying **AIRCR.PRIS**:

- If there are multiple pending exceptions, the pending exception with the lowest group priority field value takes precedence.
- If multiple pending exceptions have the same group priority field value, the pending exception with the lowest subpriority field value takes precedence.
- If multiple pending exceptions have the same group priority field value and the same subpriority field value, the pending exception with the lowest exception number takes precedence.
- If a pending Secure exception and a pending Non-secure exception both have the same group priority field value, the same subpriority field value, and the same exception number, the Secure exception takes precedence.

Note, a Secure exception requires S.

R_{KNHG}

If there are multiple pending exceptions it is IMPLEMENTATION DEFINED whether the **AIRCR.PRIGROUP** mask is applied to:

- The active tree only.
- The active tree and the pending tree.

I_{NCDS} The following is an example of exceptions with different priorities:

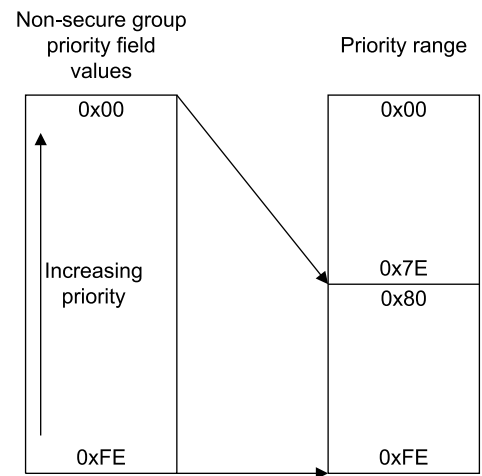
This example considers the following exceptions, that all have configurable priority numbers:

- A has the highest priority.
- B has medium priority.
- C has lowest priority.

Example sequence of events:

1. No exception is active and no priority boosting is active.
2. B is generated. The PE takes exception B and starts executing the handler for it. Exception B is now active and the current execution priority is that of B.
3. A is generated. A is higher priority, therefore A preempts B, and the PE starts executing the handler for A. Exception A is now active and the current execution priority is that of A. Exception B remains active.
4. C is generated. C has the lowest priority, therefore it is pending.
5. The PE reduces the priority of A to a priority that is lower than C. B is now the highest priority active exception, therefore the execution priority moves to that of B. The PE continues executing the handler for A at the priority of B. After completing A, the PE restarts the handler for B. After completing B, the PE takes exception C and starts executing the handler for it. C is now active and the current execution priority is that of C.

I_{XFVH} The following diagram shows an example. In this example, all 8 bits of **SHPRn_NS.PRI_n** are implemented as priority bits and **AIRCR.PRIGROUP_NS** is set to 0.



In this example, the mapping is:

SHPRn_NS.PRI_n value	Mapped to
0x00	0x80
0x02	0x81
0x04	0x82
0x06	0x83
.	.
.	.
.	.
0xFE	0xFF

In this example, Secure exceptions in the range 0x00-0x7F have priority over all Non-secure exceptions.

The extension requirements are - **M** && **S**.

I_{WPCP}

In a PE without the Main Extension but with the Security Extension, when [AIRCR.PRIS](#) is set to 1 the Non-secure exception is mapped to the lower half of the priority range, as shown in the table:

Non-secure group priority value	Mapped to
0x00	0x80
0x40	0xA0
0x80	0xC0
0xC0	0xE0

The extension requirements are - **S** && **!M**.

See also:

[B3.9 Exception numbers and exception priority numbers on page 65.](#)

[B3.30 Special-purpose mask registers, PRIMASK, BASEPRI, FAULTMASK, for configurable priority boosting on page 117.](#)

[B3.29 Hardware-controlled priority escalation to HardFault on page 116.](#)

`ExceptionPriority()`.

`ExecutionPriority()`.

`ComparePriorities()`.

`RawExecutionPriority()`.

B3.14 Secure address protection

R_{CHJX} *NS-Req* defines the Security state that the PE or DAP requests that a memory access is performed in.
 The extension requirements are - **S**.

R_{MSNJ} *NS-Attr* marks a memory access as Secure or Non-secure.
 The extension requirements are - **S**.

R_{VHRL} For PE data accesses, *NS-Req* is equal to the current Security state.
 The extension requirements are - **S**.

R_{XSPQ} For PE and DAP data accesses, *NS-Attr* is determined as follows:

NS-Req	Security attribute of the location being accessed	NS-Attr
Non-secure	Non-secure	Non-secure
	Secure	N/a
Secure	Non-secure	Non-secure
	Secure	Secure

If the *NS-Req* is Non-secure and the Security attribute of the location being accessed is Secure *NS-Attr* is not generated and an AUVIOL or INVEP exception is generated.

The extension requirements are - **S**.

R_{TDNR} For instruction fetches, *NS-Req* and *NS-Attr* are equal to the Security attribute of the location being accessed. *NS-Attr* also determines the Security state of the PE.
 The extension requirements are - **S**.

I_{NGXH} It is not possible to execute Secure code in Non-secure state, or Non-secure code in Secure state.
 The extension requirements are - **S**.

See also:

[B3.15 Security state transitions on page 82.](#)

[B11.3.4 DAP access permissions on page 245.](#)

B3.15 Security state transitions

R_{POHT} For a transition to an address in the other Security state, the following table shows when the PE changes Security state:

Current Security state	Security attribute of the branch target address	Conditions for a change in Security state
Secure	X	Change to Non-secure state if the branch was an <i>interstating branch</i> instruction, BXNS or BLXNS , with the least significant bit of its target address set to 0.
Non-secure	Secure and Non-secure callable	Change to the Secure state if both: - The branch target address contains an SG instruction which is fetched and executed. - The whole of the instruction at the branch target address is flagged as Secure and Non-secure callable.

The extension requirements are - **S**.

I_{KWMP} **SG** instructions in Secure memory are valid entry points to Secure code. They prevent Non-secure code from being able to jump to arbitrary addresses in Secure code.

The extension requirements are - **S**.

I_{WJRL} When an interstating branch is executed in Secure state, the least significant bit of the target address indicates the target Security state:

1: The target Security state is Secure.

0: The target Security state is Non-secure.

Interstating branches are **UNDEFINED** in Non-secure state.

The extension requirements are - **S**.

R_{WKXR} On transition from Secure to Non-secure state, if the least significant bit of an interstating branch is set to one, the execution of the next instruction will generate either an **INVTRAN SecureFault** or **Secure HardFault**.

The extension requirements are - **S**. Note, an **INVTRAN SecureFault** requires **M**.

R_{JKJD} On transition from Non-secure to Secure state, if there is no **SG** instruction or the whole instruction at the branch target address is not flagged as Secure and Non-secure callable the execution of the next instruction will generate either an **INVEP SecureFault** or **Secure HardFault**.

The extension requirements are - **S**. Note, an **INVTRAN SecureFault** requires **M**.

R_{XNVW} If sequential instruction execution crosses from Non-secure memory to Secure memory, then if the Secure memory entry point contains an **SG** instruction and the whole of the instruction at the Secure memory entry point is flagged as Secure and Non-secure callable, it is **CONSTRAINED UNPREDICTABLE** whether:

- The PE changes to Secure state.
- Either an **INVTRAN SecureFault** or **Secure HardFault** is generated:

The extension requirements are - **S**. Note, an **INVTRAN SecureFault** requires **M**.

R_{DWXH} When an exception is taken to the other Security state, the PE automatically transitions to that other Security state.

The extension requirements are - **S**.

See also:

[C1.4.7 Instruction set, interworking and interstating support on page 323.](#)

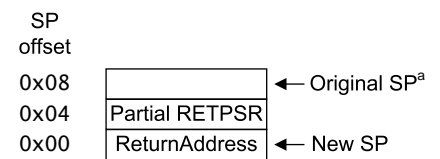
[Chapter B8 The Armv8-M Protected Memory System Architecture on page 214.](#)

B3.16 Function calls from Secure state to Non-secure state

R_{GVBB}

If a **BLXNS** interstating branch generates a change from Secure state to Non-secure state, then before the Security state change:

- The return address, which is the address of the instruction after the instruction that caused the function call, the **IPSR** value and **CONTROL.SFPA** are stored onto the current stack, as shown in the following figure. ReturnAddress[0] is set to 1 to indicate a return to the T32 instruction set state. The **IPSR** is stacked in the partial **RETPSR**, and **CONTROL.SFPA** is stacked in bit [20] of the partial **RETPSR**.



- If the PE is in Handler mode, **IPSR** has the value of 1.
- The **FNC_RETURN** value is saved in the **LR**.

The extension requirements are - **S**. Note, **CONTROL.SFPA** requires **FP**.

R_{MNOV}

Behavior is **CONSTRAINED UNPREDICTABLE** when a function call stack frame is not doubleword-aligned, and one of the following behaviors must occur:

- The instruction uses the current value of the stack pointer.
- The instruction behaves as though bits [2:0] of the stack pointer are 0b000.

The extension requirements are - **S**.

I_{KWZD}

Arm recommends that when Secure code calls a Non-secure function, any registers not passing function arguments are set to 0.

The extension requirements are - **S**.

See also:

[C1.4.7 Instruction set, interworking and interstating support on page 323.](#)

B3.17 Function returns from Non-secure state

R_{HPFG} An interstating function return begins when one of the following instructions loads a **FNC_RETURN** value into the PC:

- A **POP (multiple registers)** or **LDM** that includes loading the PC.
- An **LDR** with the PC as a destination.
- A **BX** with any register.
- A **BXNS** with any register.

On detecting a **FNC_RETURN** value in the PC:

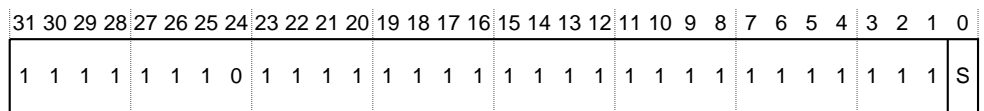
- The **FNC_RETURN** stack frame is unstacked.
- **EPSR.IT** is set to 0b00.
- The following *integrity checks on function return* are performed:
 - A check that **IPSR** is zero or 1 before the value of it is restored.
 - A check that if the stacked **IPSR** value is zero the return is in Thread mode.
 - A check that if the stacked **IPSR** value is nonzero the return is to Handler mode.

The extension requirements are - **S**.

I_{QBJN} Behavior is **CONSTRAINED UNPREDICTABLE** when a function call stack frame is not doubleword-aligned.

The extension requirements are - **S**.

R_{DWTF} The **FNC_RETURN** value is:



Bits[31:1]

This is what identifies the value as an **FNC_RETURN** value.

Bit[0], S: The function return was from:

- 0:** Secure state.
- 1:** Non-secure state.

The extension requirements are - **S**.

R_{QLJT} Any failed integrity check on function return generates a Secure INVPC UsageFault that is synchronous to the instruction that loaded the **FNC_RETURN** value into the **PC**.

The extension requirements are - **M && S**.

R_{NTNW} Any failed integrity check on function return generates a Secure HardFault that is synchronous to the instruction that loaded the **FNC_RETURN** value into the **PC**.

The extension requirements are - **S && !M**.

R_{FGNB} If **FNC_RETURN** does not fail the integrity checks then the PE behaves as follows:

- ReturnAddress bits [31:1] is written to the **PC**.
- ReturnAddress bit [0] is written to **EPSR.T**.
- The partial **RETPSR** is written to **IPSR** Exception and **CONTROL.SFPA**.

The extension requirements are - **S**. Note, **CONTROL.SFPA** requires **FP**.

R_{LNF} If the [IPSR](#) retrieved from [RETPSR](#) is not supported by the PE the value is UNKNOWN.

The extension requirements are - [S](#).

I_{KBXQ} Any Secure INVPC UsageFault, Secure HardFault, or INVSTATE UsageFault generated on [FNC_RETURN](#) are subject to the rules in respect of escalation of faults and potentially [Lockup](#).

The extension requirements are - [S](#).

See also:

[B3.16 Function calls from Secure state to Non-secure state](#) on page 84.

[B3.29 Hardware-controlled priority escalation to HardFault](#) on page 116.

[B3.31 Lockup](#) on page 119.

B3.18 Exception handling

- R_{XGKT}** In the absence of a specific requirement to take an exception, the architecture requires that pending exceptions are taken within finite time.
- R_{KFRF}** If an exception was pending but is changed to not pending before it is taken, then the architecture permits the exception to be taken but does not require that the exception is taken. If the exception is taken it must be taken before the first [Context synchronization event](#) after the exception was changed to not pending.
- R_{YFHR}** An exception that does not cause [lockup](#) sets both:
- The pending bit of its handler, or the pending bit of the HardFault handler, to 1.
 - The associated fault status information.
- R_{VLDB}** When a pending exception has a lower group priority value than current execution, including accounting for any priority adjustment by [AIRCR.PRIS](#), the pending exception preempts current execution.
- R_{WBND}** Preemption of current execution causes the following basic sequence:
1. R0-R3, R12, [LR](#), [RETSPSR](#), including [CONTROL.SFPA](#), are stacked.
 2. The return address is determined and stacked.
 3. Optional stacking of Floating-point context, which might be any one of the following:
 - No stacking or preservation of the Floating-point context.
 - Stacking the basic Floating-point context.
 - Stacking the basic Floating-point context and the additional Floating-point context.
 - Activation of Lazy Floating-point state preservation.
 4. [LR](#) is set to [EXC_RETURN](#).
 5. Optional clearing of Floating-point registers, depending on the Security state transition.
 6. The following flags are also cleared:
 - IT State is cleared, if the Main Extension is implemented.
 - [CONTROL.FPCA](#) is cleared, if the Floating-point Extension is implemented.
 - [CONTROL.SFPA](#) is cleared, if the Floating-point Extension and the Security Extension are implemented.
 7. The exception to be taken is chosen, and [IPSR](#) Exception is set accordingly. The setting of [IPSR](#) Exception to a nonzero value causes the PE to change to Handler mode.
 8. [CONTROL.SPSEL](#) is set to 0, to indicate the selection of the main stack, dependent on the Security state being targeted.
 9. The pending bit of the exception to be taken is set to 0. The active bit of the exception to be taken is set to 1.
 10. The Security state is changed to the Security state of the exception that is being activated.
 11. The registers are cleared, depending on the transition of the Security state. The registers are divided between the caller and callee registers. If the Security state transition is from Secure to Non-secure state, all the registers are cleared to 0. In all other cases, the caller registers are set to an UNKNOWN value and the callee registers remain unchanged and are not stacked.
 12. [EPSR.T](#) is set to bit[0] of the exception vector for the exception to be taken.

13. The PC is set to the exception vector for the exception to be taken.

Note, some steps might require additional extensions.

R_{NJVF}

When, during exception entry, the target Security state of an exception differs from the Security state of the memory the exception vector targets:

- An INVEP SecureFault is generated if the exception is Non-secure and the exception vector targets Secure memory.
 - The INVEP SecureFault can be avoided if the exception is associated with Non-secure state and is targeting an SG instruction that is located in memory that is Secure and Non-secure callable.
- An INVTRAN SecureFault is generated if the exception is Secure and the exception vector targets Non-secure memory.

The extension requirements are - S. Note, an INVEP or INVTRAN SecureFault requires M.

R_{QLHB}

The return address is one of the following:

- For a synchronous exception, other than an SVCcall exception and a SVC instruction that escalates to HardFault, the address of the instruction that caused the exception.
- For an asynchronous exception, the address of the next instruction in the program order.
- For an SVCcall exception and a SVC instruction that escalates to HardFault, the address of the next instruction in the program order.

R_{XKDD}

The least significant bit of the return address from an exception is RES0.

See also:

[B3.10 Exception enable, pending, and active bits on page 68.](#)

[B3.13 Priority model on page 77.](#)

[B3.19 Exception entry, context stacking on page 89.](#)

[B3.20 Exception entry, register clearing after context stacking on page 97.](#)

[B3.28 Vector tables on page 114.](#)

[B3.21 Stack limit checks on page 98.](#)

[B3.24 Exceptions during exception entry on page 105.](#)

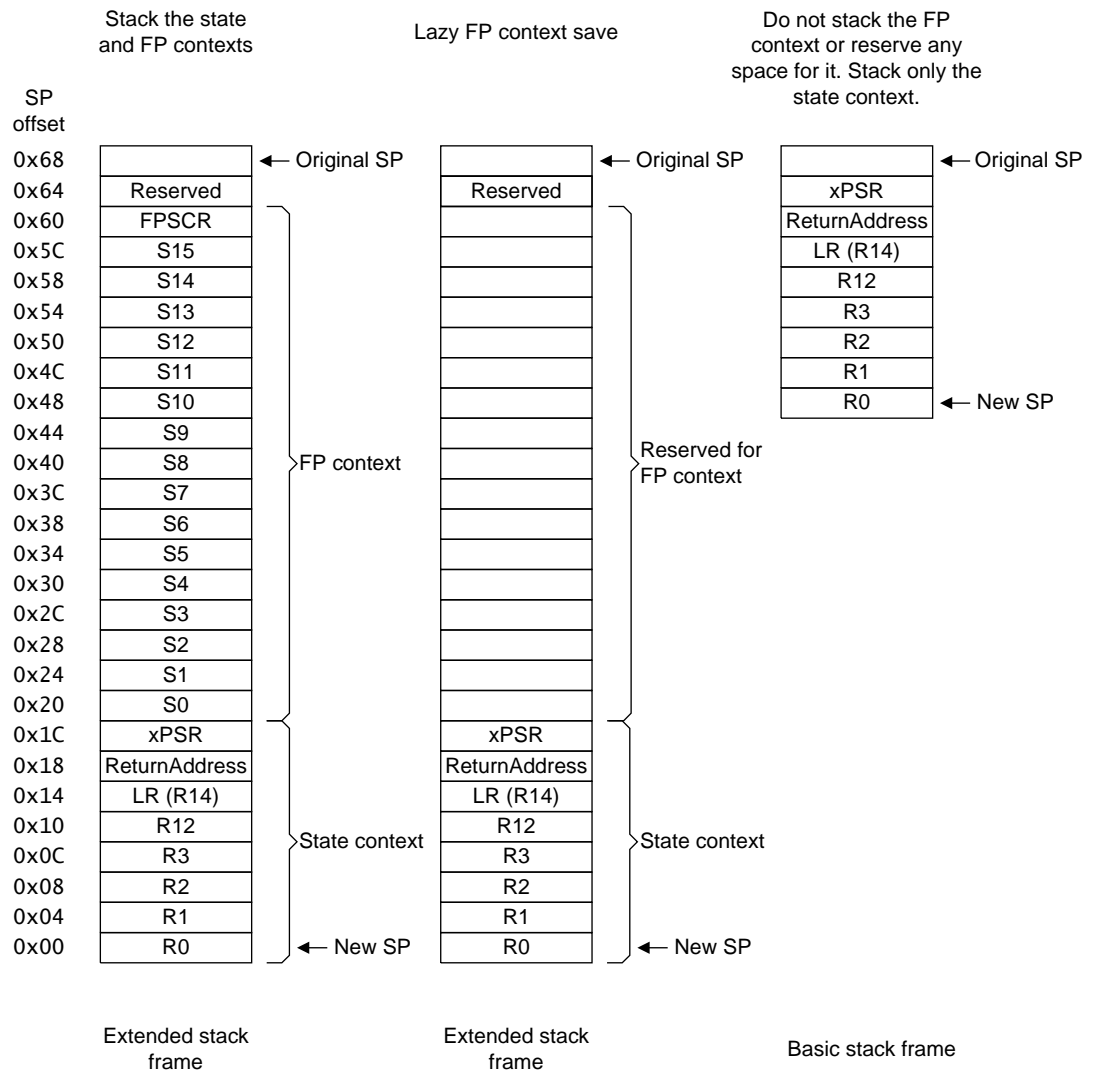
B3.19 Exception entry, context stacking

R_{PWWG} On taking an exception, the PE hardware saves *state context* onto the stack that the SP register points to. The state context that is saved is eight 32-bit words:

- RETPSR.
- ReturnAddress.
- LR.
- R12.
- R3-R0.

R_{PTRL} In a PE without the Security Extension but with the Floating-point Extension, on taking an exception, the PE hardware saves state context onto the stack that the SP register points to. If CONTROL.FPCA is 1 when the exception is taken, then in addition to the state context being saved, there are the following possible modes for the *Floating-point context*:

- Stack the Floating-point context.
- Reserve space on the stack for the Floating-point context. This is called *lazy Floating-point context preservation*.



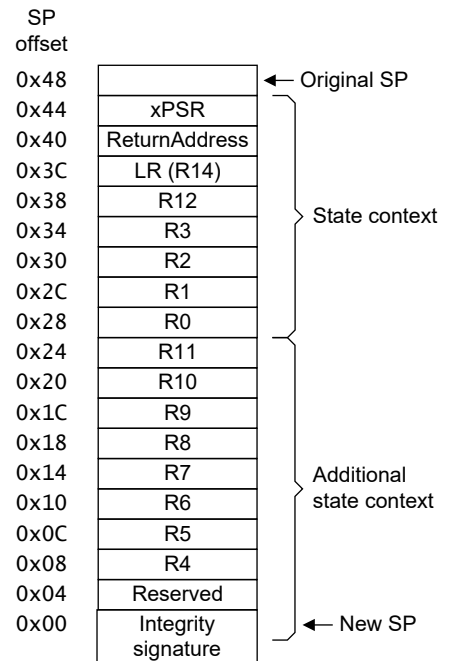
The extension requirements are - !S. Note, FP is required for the extended stack frame.

R_{PLHM}

In a PE with the Security Extension, on taking an exception, the PE hardware:

1. Saves state context onto the stack that the **SP** register points to.
2. If exception entry requires a transition from Secure state to Non-secure state, the PE hardware extends the stack frame and also saves *additional state context*.

Exception taken from Secure state with Stacking of additional state context



The extension requirements are - **S**.

- R_{BLQS}** If a Secure exception is taken from a Secure context of execution, it is IMPLEMENTATION DEFINED whether:
- The additional state context is not stacked, and **EXC_RETURN.DCRS** is set to 1.
 - The additional state context is stacked and **EXC_RETURN.DCRS** is set to 0.

The extension requirements are - **S**.

- I_{KJRL}** If a higher priority Secure exception occurs during exception entry after the PE has begun stacking the additional state context, but before any exception handler has started execution, in preparation to take a Non-secure exception the PE might:
- Discard the stacking of the additional state context.
 - Complete the stacking of the additional state context and the **EXC_RETURN.DCRS** is set to 0.

The extension requirements are - **S**.

- R_{DHPD}** In a PE with the Security Extension and the Floating-point Extension, on taking an exception from:

Non-secure state

Behavior is the same as a PE without the Security Extension but with the Floating-point Extension.

Secure state when CONTROL.FPCA is 0

Behavior is the same as for a PE with the Security Extension but without the Floating-point Extension.

Secure state when CONTROL.FPCA is 1

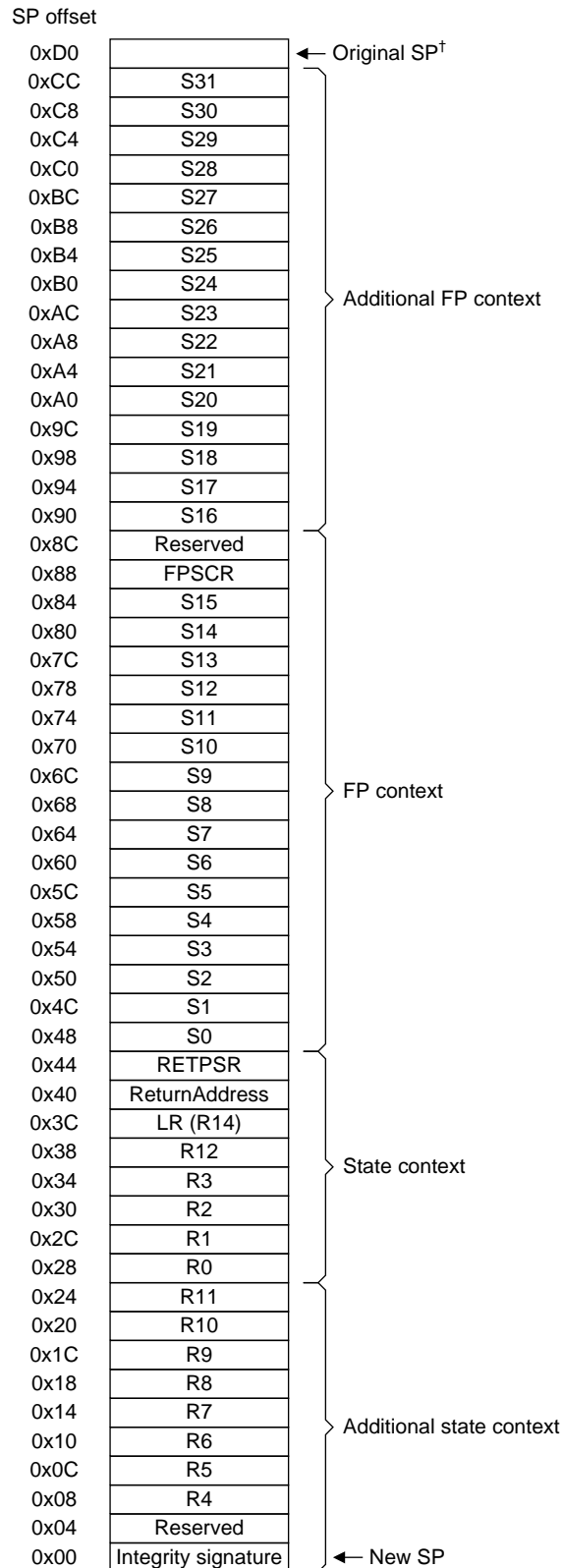
The PE hardware:

1. Saves state context onto the stack that the [SP](#) register points to.
2. If [FPCCR_S.TS](#) is 0 or the background state is Non-secure when the exception is taken, the PE hardware either stacks the Floating-point context or when lazy state preservation is active reserves space on the stack for the Floating-point context.

If [FPCCR_S.TS](#) is 1 and the background state is Secure state when the exception is taken, the PE hardware either stacks both the Floating-point context and additional Floating-point context, or when lazy state preservation is active reserves space on the stack for both the Floating-point context and additional Floating-point context.

3. If exception entry is to Non-secure state, including when a higher priority derived or late-arriving exception targeting Secure state occurs, the PE hardware extends the stack frame, and also saves the additional state context. The PE also performs the exception handling steps common to exception entry.

The following figure shows PE stacking behavior when [CONTROL.FPCA](#) is 1, [FPCCR_S.TS](#) is 1 (and both the Floating-point context and additional Floating-point context is stacked), and exception entry is to Non-secure state and the background state is Secure state:



[†] Or at offset 0xD4 if at a word-aligned but not doubleword-aligned address.

The extension requirements are - **S && FP**.

R_{BKVD} On an exception, the **RETPSR** value that is stacked is all the following:

- The **APSR**, **IPSR**, and **EPSR**.
- **CONTROL.SFPA**, in **RETPSR**[20] if the background state is Secure state.

In addition, on an exception, the PE uses **RETPSR.SPREALIGN** to indicate whether the PE realigned the stack to make it doubleword-aligned:

0: The PE did not realign the stack.

1: The PE realigned the stack.

Note, **CONTROL.SFPA** requires **S && FP**.

R_{QDKQ} When stacking the context on exception entry, full descending stacks are used.

R_{PWBW} In a PE with the Floating-point Extension:

- Because setting **FPCCR.ASPEN** to one causes the PE to automatically set **CONTROL.FPCA** to 1 on the execution of a floating-point instruction, setting **FPCCR.ASPEN** to 1 means that the PE hardware automatically either:
 - Stacks Floating-point context on taking an exception.
 - Uses *lazy Floating-point context preservation* on taking an exception.

If **CONTROL.FPCA** == 1, it is **FPCCR.LSPEN** that determines whether the PE hardware performs stacking or lazy Floating-point preservation:

0 : The PE hardware automatically stacks Floating-point context on taking an exception. In a PE that also includes the Security Extension, if **FPCCR.S.TS** == 1 and the background state is Secure state, the hardware stacks the *additional Floating-point context* and the Floating-point context.

1: The PE hardware uses lazy Floating-point context preservation on taking an exception, and sets all of:

- The **FPCAR**, to point to the reserved S0 stack address.
- **FPCCR.LSPACT** to 1.
- **FPCCR.{USER, THREAD, HFREADY, MMRDY, BFRDY, SFRDY, MONRDY, UFRDY}**, to record the permissions and fault possibilities to be applied to any subsequent Floating-point context save.

In a PE that also includes the Security Extension, if **FPCCR.S.TS** is 1 and the background state is Secure state, the hardware reserves space on the stack for both the Floating-point context and the *additional Floating-point context*. Otherwise, the hardware only reserves space on the stack for the Floating-point context.

The extension requirements are - **FP**. Note, space is reserved for both the Floating-point context and the *additional Floating-point context* if the Security Extension is implemented.

R_{GHDJ} Execution of a floating-point instruction while **FPCCR.LSPACT** == 1 indicates that lazy Floating-point context preservation is active.

The extension requirements are - **FP**.

R_{FTZK} If an attempt is made to execute a floating-point instruction while lazy Floating-point context preservation is active, the access permissions that **CPACR** and **NSACR** define are checked against the context that activated lazy Floating-point context preservation, in addition to the checks defined in **FPCCR**.

- If no permission violation is detected, the PE:

1. Saves Floating-point context to the reserved area on the stack, as identified by the `FPCAR`.
 2. Saves the additional Floating-point context if `FPCCR.TS` and `FPCCR.S == 1`.
 3. Sets `FPCCR.LSPACT` to 0 to indicate that lazy Floating-point context preservation is no longer active.
 4. If the instruction targets Non-secure state the registers are set to an UNKNOWN value. If the instruction targets Secure state the registers are cleared.
 5. Processes the floating-point instruction.
- If a permission violation is detected, the PE generates a NOCP UsageFault and does not save Floating-point context to the reserved area on the stack.
 - If there is a Security violation or other exception on context stacking the PE will take that exception if the priority of the exception is lower than the execution priority.

The extension requirements are - *FP*.

R_{LGNS} When the following conditions are met on exception entry, the PE generates a Secure NOCP UsageFault, skips all Floating-point register saving, clearing or lazy-state preservation activation and does not allocate space on the stack for Floating-point context:

- `CONTROL.FPCA == 1`.
- `NSACR.CP10` is 0.
- The `Background state` is Non-secure state.

The extension requirements are - *FP && S*.

R_{QLGM} A NOCP UsageFault takes precedence over UNDEFINSTR faults for all instructions that fall into the range described by the `IsCPIInstruction()` function, except for the following instructions:

- `VLLDM`.
- `VLSTM`.

Note, For further information see the instruction descriptions.

R_{RXDG} On taking a NOCP UsageFault any state context and additional state context must be saved.

R_{TLGR} A PE is permitted not to save state context and additional state context to the stack if a derived exception is taken on pushing the state context or additional context to the stack.

R_{KMBN} If lazy Floating-point context preservation or floating-point context stacking is activated, as indicated by `FPCCR.S.S` when `FPCCR.LSPACT` is already set to 1, the PE generates an LSERR SecureFault. The floating-point context, and the additional context, are not stacked and the floating-point registers are not cleared.

The extension requirements are - *FP && S*.

R_{FVTL} The value in `CONTROL.SFPA` is set automatically by hardware on any of the following events:

- An `SG` instruction fetched from secure memory and executed in Non-secure state clears `CONTROL.SFPA` to 0.
- A `BXNS` instruction that causes a transition from Secure state to Non-secure state clears `CONTROL.SFPA` to 0.
- A `BLXNS` instruction that causes a transition from Secure state to Non-secure state preserves the value in `CONTROL.SFPA` in the `FNC_RETURN` stack frame and then clears `CONTROL.SFPA` to 0.
- A valid instruction that loads `FNC_RETURN` into the PC sets `CONTROL.SFPA` to the value retrieved from the `FNC_RETURN` payload.
- `CONTROL.SFPA` is saved and restored on exception entry or return in the `RETPSR` value in the stack frame.
- Exception entry, including tail chaining, clears `CONTROL.SFPA` to 0.

- If the value of `FPCCR.ASPEN` is one, then any floating-point instruction (excluding `VLLDM` and `VLSTM`) executed in Secure state sets the value of `CONTROL.SFPA` to one. If the value of `FPCCR.ASPEN` is one and the value of `CONTROL.SFPA` is zero when a floating-point instruction is executed in the Secure state, the `FPSCR` value is taken from the values set in `FPDSCR`.

The extension requirements are - *FP* && *S*.

`I_GJGL`

To ensure future compatibility Arm recommends that the value used to seal the top of the stack is `0xFEFE5EDA5`. This value has the following properties:

- It is not a valid `FNC_RETURN` or `EXC_RETURN` value.
- It is not the integrity signature used to secure the bottom of the stack frame and cannot be used to inadvertently mark the stack as containing a valid exception stack frame.
- The value starts with `0xF` and is therefore not a valid instruction address and will result in a fault if interpreted as a `FNC_RETURN` stack frame.

`I_CNTRN`

The order in which registers are saved to the stack during exception entry or during lazy state preservation is not architected.

Note, FP required of lazy state preservation.

See also:

[B3.8 Stack pointer on page 63.](#)

[B3.20 Exception entry, register clearing after context stacking on page 97.](#)

[B3.23 Integrity signature on page 104.](#)

`PushStack()`.

B3.20 Exception entry, register clearing after context stacking

R_{DJRX}

On exception entry:

- The PE hardware sets R0-R3, R12, **APSR**, and **EPSR** to an UNKNOWN value after it has pushed state context to the stack.
- The PE hardware sets S0-S15 and the **FPSCR** to an UNKNOWN value after it has pushed Floating-point context to the stack.

The extension requirements are - !S. Note, FP is required for S0-S15 and FPSCR.

R_{SNDB}

On exception entry, including **tail-chaining**, the PE sets:

- R0-R3, R12, **APSR**, and **EPSR** to:
 - Unless otherwise stated, an UNKNOWN value if the exception is taken to Secure state.
 - Zeros.
- If the background state was Secure and the exception targets the Secure state and **EXC_RETURN.DCRS** == 0 then R4 to R11 become UNKNOWN.
- If the background state was Secure and the exception targets Non-secure state then R4 to R11 are set to zeros.

Otherwise the register values are not changed.

The extension requirements are - S.

R_{JWBK}

Register clearing behavior after context stacking is as follows:

- If **FPCCR.S.TS** is 0 when the Floating-point context is pushed to the stack, S0-S15 and the **FPSCR** are set to an UNKNOWN value after stacking.
- If **FPCCR.S.TS** is 1 when the Floating-point context and additional Floating-point context are both pushed to the stack, S0-S31 and the **FPSCR** are set to zero after stacking.

The extension requirements are - FP && S.

See also:

[B3.19 Exception entry, context stacking on page 89.](#)

[B3.26 Tail-chaining on page 108.](#)

B3.21 Stack limit checks

R_{PCRT} A PE that does not implement the Main Extension, and does not implement the Security Extension does not implement stack-limit checking.

The extension requirements are - !M && !S.

R_{NHBX} In a PE without the Main Extension but with the Security Extension, there are two stack limit registers in Secure state for the purposes of stack-limit checking.

Security state	Stack	Stack limit registers
Secure	Main	MSPLIM_S
	Process	PSPLIM_S

The extension requirements are - S && !M.

R_{JPFX} In a PE with the Main Extension but without the Security Extension, there are two stack limit registers:

Stack	Stack limit registers
Main	MSPLIM
Process	PSPLIM

The extension requirements are - M && !S.

R_{XQDS} In a PE with the Main Extension and the Security Extension, there are four stack limit registers:

Security state	Stack	Stack limit registers
Secure	Main	MSPLIM_S
	Process	PSPLIM_S
Non-secure	Main	MSPLIM_NS
	Process	PSPLIM_NS

The extension requirements are - M && S.

I_{KDPG} A stack can descend to its stack limit value. Any attempt to descend the stack further than its stack limit value is a violation of the stack limit.

R_{TCXN} xSPLIM_x[2:0] are treated as RES0, so that all stack pointer limits are always guaranteed to be doubleword-aligned. Bits [31:3] of the xSPLIM_x registers are writable.

R_{DKSR} Stack limit checks are performed during the creation of a stack frame for all of the following:

- Exception entry.
- [Tail-chaining](#) from a Secure to a Non-secure exception.
- A function call from Secure code to Non-secure code.

Note, Secure exceptions and secure code require S.

R_{ZLZG} On a violation of a stack limit during either exception entry or [tail-chaining](#):

- In a PE with the Main Extension, a synchronous STKOF UsageFault is generated. Otherwise, a HardFault is generated.
- The stack pointer is set to the stack limit value.
- Push operations to addresses below the stack limit value are not performed.
- It is IMPLEMENTATION DEFINED whether push operations to addresses equal to or above the stack limit value are performed.

Note, A UsageFault requires M.

- R_{CCSC}** On a violation of a Secure stack limit during a function call:
- In a PE with the Main Extension, a synchronous STKOF UsageFault is generated. Otherwise, a Secure HardFault is generated.
 - Push operations to addresses below the stack limit value are not performed.
 - It is IMPLEMENTATION DEFINED whether push operations to addresses equal to or above the stack limit value are performed.

The extension requirements are - **S**. Note, A UsageFault requires M.

- R_{GGRH}** Unstacking operations are not subject to stack limit checking.

- R_{YVWT}** Updates to the stack pointer by the following instructions are subject to stack limit checking:

- ADD (SP plus immediate).
- ADD (SP plus register).
- SUB (SP minus immediate).
- SUB (SP minus register).
- BLX, BLXNS.
- LDC, LDC2 (immediate).
- LDM, LDMIA, LDMFD.
- LDMDB, LDMEA.
- LDR (immediate).
- LDR (literal).
- LDR (register).
- LDRB (immediate).
- LDRD (immediate).
- LDRH (immediate).
- LDRSB (immediate).
- LDRSH (immediate).
- MOV (register)
- POP (multiple registers).
- PUSH (multiple registers).
- VPOP.
- VPUSH.
- STC, STC2
- STM, STMIA, STMEA.
- STMDB, STMFD.
- STR (immediate).
- STRB (immediate).
- STRD (immediate).
- STRH (immediate).
- VLDM.
- VSTM.

Updates to the stack pointer by the **MSR** instruction targeting **SP_NS** are subject to stack limit checking. Updates to the stack pointer and stack pointer limit by any other **MSR** instruction are not subject to stack limit checking.

LDR instructions write to two registers, the address register and the destination register. The stack limit check is only carried out against the address register. Updates to the stack pointer by the **LDR** instructions are only subject to stack limit checking if the stack pointer is the address register.

For all other instructions that can update the stack pointer and stack pointer limit, it is IMPLEMENTATION DEFINED whether stack limit checking is performed.

- I_{BJHX}** When an instruction updates the stack pointer, if it results in a violation of the stack limit, it is the modification of the stack pointer that generates the exception, rather than an access that uses the out-of-range stack pointer.
- I_{RRDX}** [CCR.STKOFHFNMIGN](#) controls whether stack limit violations are IGNORED while executing at a requested execution priority that is negative.
- R_{XCQL}** It is UNKNOWN whether a stack limit check is performed on any use of the SP marked as UNPREDICTABLE.
- R_{QFPF}** A write to the current stack pointer by an instruction subject to stack limit checking with a value less than the stack limit will generate a STKOF UsageFault.
- I_{DSJN}** There is no architectural requirement for stack limit checking to be carried out on exception return as the current stack pointer will only increment and will not decrement.
- R_{QMRP}** If an instruction attempts to make an access below the stack limit, it is UNKNOWN whether a store performing a writeback to the current Stack Pointer will generate a STKOF UsageFault where the value written to the current stack pointer is greater than the stack limit.
The extension requirements are - [M](#) || [S](#).
- R_{CMBW}** When a STKOF UsageFault is generated:
- No accesses below the stack limit will be performed.
 - It is UNKNOWN whether an access above the stack limit will be performed.

See also:

[B3.8 Stack pointer on page 63.](#)

[B3.26 Tail-chaining on page 108.](#)

B3.22 Exception return

- R_{KPSS}** The PE begins an exception return when both of the following are true:
- The PE is in Handler mode.
 - One of the following instructions loads an **EXC_RETURN** value, `0xFFXXXXXX`, into the **PC**:
 - A **POP (multiple registers)** or **LDM** that includes loading the **PC**.
 - An **LDR** with the **PC** as a destination.
 - A **BX** with any register.
 - A **BXNS** with any register.

When both of these are true, then on detecting an **EXC_RETURN** value in the **PC**, the PE unstacks the exception stack frame and resumes execution of the unstacked context.

If an **EXC_RETURN** value is loaded into the **PC** by an instruction other than those listed, or from the vector table, the value is treated as an address.

If an **EXC_RETURN** value is loaded into the **PC** when the PE is in Thread mode, the value is treated as an address.

- R_{TXDW}** Behavior is UNPREDICTABLE if **EXC_RETURN.FType** is 0 and the Floating-point Extension register file is not implemented.

- R_{TNSK}** Behavior is UNPREDICTABLE if **EXC_RETURN[23:7]** are not all 1 or if bit[1] is not 0.

- R_{XLCP}** Behavior is UNPREDICTABLE if any of the following are true and the Security Extension is not implemented:
- **EXC_RETURN.S** is 1.
 - **EXC_RETURN.DCRS** is 0.
 - **EXC_RETURN.ES** is 1.

- R_{LLBT}** The following integrity checks on exception return are performed on every exception return:
1. In a PE with the Security Extension, the integrity check that is called the **EXC_RETURN.ES validation check**, as follows:
 - If the PE was in Non-secure state when **EXC_RETURN** was loaded into the **PC** and either **EXC_RETURN.DCRS** is 0 or **EXC_RETURN.ES** is 1, an INVER SecureFault is generated and the PE sets **EXC_RETURN.ES** to 0. In a PE without the Main Extension a Secure HardFault is generated.
 2. A check that the exception number being returned from, as held in the **IPSR**, is shown as active in the **SHCSR** or **NVIC_IABRn**. If this check fails:
 - In a PE with the Main Extension, an INVPC UsageFault is generated. If the PE includes the Security Extension, the INVPC UsageFault targets the Security state that the exception return instruction was executed in.
 - In a PE without the Main Extension, a HardFault is generated.
 3. A check that if the return is to Thread mode, the value that is restored to the **IPSR** from the **RETPSR** is zero, or that if the return is to Handler mode, the value that is restored to the **IPSR** from the **RETPSR** is nonzero. If this check fails:
 - In a PE with the Main Extension, an INVPC UsageFault is generated. If the PE includes the Security Extension, the INVPC UsageFault targets the Background state.

- In a PE without the Main Extension, a HardFault is generated.
4. If the PE includes the Security Extension, the HardFault targets the Security state that `EXC_RETURN.S` specifies. If `AIRCR.BFHFNMINs` is 0 the HardFault targets Secure state, if `AIRCR.BFHFNMINs` is 1 the exception targets the Security state the exception was returned from.

Note, some steps require additional extensions, as listed in the rule.

<code>R_{HXSr}</code>	When returning from Non-secure state, <code>EXC_RETURN.ES</code> is treated as zero for all purposes other than raising the <code>INVER</code> integrity check. <i>The extension requirements are - S.</i>
<code>R_{DQLL}</code>	On returning from Non-secure state, if <code>EXC_RETURN.ES</code> causes an <code>INVER</code> integrity check failure, the subsequent <code>EXC_RETURN.DCRS</code> bit that is presented in the <code>LR</code> on entry to the next exception is permitted to be UNKNOWN. <i>The extension requirements are - S.</i>
<code>I_{TLXJ}</code>	Arm recommends that the subsequent <code>EXC_RETURN.DCRS</code> bit that is presented in the <code>LR</code> on entry to the next exception is not UNKNOWN.
<code>R_{JMJC}</code>	After the <code>EXC_RETURN.ES</code> validation check has been performed on an exception return: <ul style="list-style-type: none"> • If <code>EXC_RETURN.ES</code> is 1, <code>EXC_RETURN.SPSEL</code> is written to <code>CONTROL_S.SPSEL</code>. • If <code>EXC_RETURN.ES</code> is 0, <code>EXC_RETURN.SPSEL</code> is written to <code>CONTROL_NS.SPSEL</code>. <i>The extension requirements are - S.</i>
<code>R_{RPGL}</code>	On an exception return that successfully returns to the Background state, with no <code>tail-chaining</code> or failed integrity checks, the Security state is set to <code>EXC_RETURN.S</code> . <i>The extension requirements are - S.</i>
<code>I_{CTWL}</code>	In a PE with the Security Extension, after a successful exception return to the Background state, the PE is in the correct Security state before the next instruction from the background code is executed. This means that in the case where the Background state is Secure state, there is no need for an <code>SG</code> instruction at the exception return address. <i>The extension requirements are - S.</i>
<code>I_{RQVB}</code>	In a PE with the Floating-point Extension register file, on exception entry: <ol style="list-style-type: none"> 1. <code>EXC_RETURN.FType</code> is saved as the inverse of <code>CONTROL.FPCA</code>. 2. <code>CONTROL.FPCA</code> is then cleared to 0 if it was 1. On exception return, the inverse of <code>EXC_RETURN.FType</code> is written to <code>CONTROL.FPCA</code> . <i>The extension requirements are - FP.</i>
<code>R_{CGML}</code>	When the following conditions are met on exception return, the PE hardware sets S0-S15 and the <code>FPSCR</code> to 0: <ul style="list-style-type: none"> • <code>CONTROL.FPCA</code> is 1. • <code>FPCCR.CLRONRET</code> is 1. • If the PE implements the Security Extension <code>FPCCR_S.LSPACT</code> is 0. If the PE implements the Security Extension and all these fields are 1 on exception return, the PE generates an LSERR SecureFault instead. <i>The extension requirements are - FP. Note, a SecureFault requires S.</i>

- I_{RHNB}** [IsCPEnabled\(\)](#) indicates the prioritization if the access is blocked by multiple registers.
- R_{XNNG}** When the following conditions are met on exception return, the PE generates an LSERR SecureFault:
- [EXC_RETURN.FType](#) is 0.
 - The stack might contain Secure Floating-point context or Secure lazy floating-point context, that would be unstacked on return. That is, [FPCCR_S.LSPACT](#) is 1.
 - The return is to Non-secure state.
- The extension requirements are - **FP** && **S**.*
- R_{VGGF}** A check of [FPCCR_S.LSPACT](#), [CPACR.CP10](#), and the relevant fields in [NSACR](#) and [CPPWR](#) is undertaken prior to unstacking of the floating-point registers.
- The extension requirements are - **FP**.*
- R_{GDVT}** The floating-point registers are not modified if the checks prior to unstacking fail.
- The extension requirements are - **FP**.*
- R_{HNNW}** If the PE abandons unstacking of the floating-point registers to [tail-chain](#) into another exception, then if the Security Extension is implemented, the PE clears to zero any floating-point registers that would have been unstacked.
- The extension requirements are - **FP** && **S**.*
- R_{LMNG}** If the PE abandons unstacking of the floating-point registers to [tail-chain](#) into another exception, then if the Security Extension is not implemented, the floating-point registers that would have been unstacked become UNKNOWN.
- The extension requirements are - **FP** && **!S**.*
- R_{HRJH}** Following completion of the requirements of the [EXC_RETURN](#) the PE returns to execution and the following occurs:
- The registers pushed to the stack as part of the exception entry are restored from the stack frame (in accordance with the [EXC_RETURN](#) flags).
 - [APSR](#), [EPSR](#), and [IPSR](#) are restored from [RETPSR](#).
 - The [PC](#) is set to `ReturnAddress [31:1]: '0'`.
 - Bit[0] of the `ReturnAddress` is discarded.
- I_{GTDP}** The order in which registers, including floating-point registers, are restored from the stack is not architected.

See also:

[B3.18 Exception handling on page 87.](#)

[ExceptionReturn\(\)](#)

B3.23 Integrity signature

R_{PHBP} In a PE with the Floating-point Extension register file, the integrity signature value is:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	0	1	1	1	1	1	0	1	0	0	0	0	1	0	0	1	0	0	1	0	1	1	0	1	SFTC

Stack Frame Type Check ↴

In a PE with the Floating-point Extension, when returning from a Non-secure exception to Secure state, if the unstacked integrity signature does not match this value, including if SFTC does not match [EXC_RETURN.FType](#), a SecureFault is generated.

The extension requirements are - [S](#) && [FP](#).

R_{MVKS} In a PE without the Floating-point Extension register file, the integrity signature value is:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	0	1	1	1	1	1	0	1	0	0	0	0	1	0	0	1	0	0	1	0	1	1	0	1	1

- In a PE with the Main Extension, when returning from a Non-secure exception to Secure state, if the unstacked integrity signature does not match this value, a SecureFault is generated.
- In a PE without the Main Extension, when returning from a Non-secure exception to Secure state, if the unstacked integrity signature does not match this value, a Secure HardFault is generated.

The extension requirements are - [S](#) && [!FP](#). Note, a SecureFault requires [M](#).

I_{FFTS} The integrity signature is an XN address. When performing a function return from Non-secure code, if the integrity signature value is restored to the PC as the function return address, a MemManage fault, if the Main Extension is implemented, or a HardFault, in an implementation without the Main Extension, is generated when the PE attempts execution.

The extension requirements are - [S](#).

See also:

[B3.19 Exception entry, context stacking on page 89.](#)

[B3.22 Exception return on page 101.](#)

B3.24 Exceptions during exception entry

I_{LBGQ} During exception entry exceptions can occur, for example asynchronous exceptions, or the exception entry sequence itself might cause an exception, for example a MemManage fault on the push to the stack.

Any exception that occurs during exception entry is a late-arriving exception, and:

- The exception that caused the original entry sequence is the *original exception*.
- The priority of the code stream running at the time of the original exception is the *preempted priority*.

When the exception entry sequence itself causes an exception, the latter exception is a *derived exception*.

The following mechanism is called *late-arrival preemption*. The PE takes a late-arriving exception during an exception entry if the late-arriving exception is higher priority, including accounting for any priority adjustment by [AIRCR.PRIS](#).

R_{NMTT} In late-arrival preemption:

- The late-arriving exception uses the exception entry sequence started by the original exception. The original exception remains pending.
- The PE takes the original exception after returning from the late-arriving exception.
- The PE ignores non-terminal faults on taking a derived exception on late-arrival preemption.

The pseudocode [DerivedLateArrival\(\)](#) describes this.

R_{MRTT} For Derived exceptions, late-arrival preemption is mandatory.

R_{BXTB} For late-arriving asynchronous exceptions, it is IMPLEMENTATION DEFINED whether late-arrival preemption is used. If the PE does not implement late-arrival preemption for late-arriving asynchronous exceptions, late-arriving asynchronous exceptions become pending.

R_{GVHV} If a higher priority late-arriving Secure exception occurs during entry to a Non-secure exception when the Background state is Secure, it is IMPLEMENTATION DEFINED whether:

- The stacking of the additional state context is rolled back.
- The stacking of the additional state context is completed and [EXC_RETURN](#) is set to 0.

The extension requirements are - [S](#).

R_{GDNT} If the group priority value of a derived exception is higher than or equal to the preempted priority:

- If the derived exception is a DebugMonitor exception, it is IGNORED.
- Otherwise, the PE escalates the derived exception to HardFault or [Lockup](#) if the HardFault cannot be taken due to the current priority.

Note, a DebugMonitor Exception requires the DebugMonitor exception.

I_{NJCW} The architecture does not specify the point during exception entry at which the PE recognizes the arrival of an asynchronous exception.

See also:

[B3.9 Exception numbers and exception priority numbers on page 65.](#)

[B3.13 Priority model on page 77.](#)

[B3.18 Exception handling on page 87.](#)

[B3.26 Tail-chaining on page 108.](#)

B3.25 Exceptions during exception return

- I_{KXPV}** During exception return exceptions can occur, for example asynchronous exceptions, or the exception return might itself cause an exception.
Any exception that occurs during exception return is a *late-arriving exception*.
When the exception return sequence itself causes an exception, the latter exception is a derived exception.
- R_{TRFM}** When a late-arriving exception during exception return has a lower priority value than the priority being returned to, the PE takes the late-arriving exception by using [tail-chaining](#).
- I_{MBNG}** The architecture does not specify the point during exception return at which the PE recognizes the arrival of an asynchronous exception. If a PE recognizes an asynchronous exception after an exception return has completed, there is no opportunity to [tail-chain](#) the asynchronous exception.
- R_{MJDN}** If the priority value of a derived exception during exception return is equal to or higher than the priority being returned to:
- If the derived exception is a DebugMonitor exception, the PE ignores the derived exception.
 - Otherwise, the PE escalates the derived exception to HardFault and the escalated exception is [tail-chained](#).
- Note, a DebugMonitor Exception requires the DebugMonitor exception.*
- R_{DHFK}** If the priority value of a derived exception during exception return, after priority escalation if appropriate, is a lower priority value than the execution priority being returned to, the PE uses [tail-chaining](#) to take the derived exception.
- I_{PXRQ}** If the PE cannot escalate a derived exception to HardFault because the current execution priority cannot be preempted the PE will enter `glos:Lockup]Lockup`. Arm recommends that entry into Lockup is treated as fatal and requiring the PE to be reset. On reset any saved context or additional context is discarded.

See also:

[B3.9 Exception numbers and exception priority numbers on page 65.](#)

[B3.13 Priority model on page 77.](#)

[B3.22 Exception return on page 101.](#)

[B3.26 Tail-chaining on page 108.](#)

[B3.31 Lockup on page 119](#)

[DebugMonitor exception.](#)

B3.26 Tail-chaining

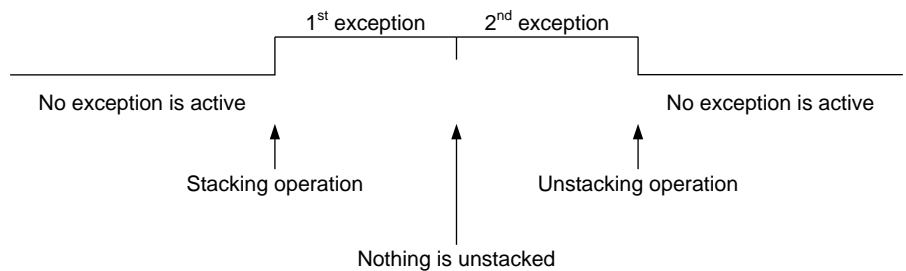
R_{FKXX} *Tail-chaining* behavior is as follows:

On detecting an **EXC_RETURN** value in the **PC**, if there is a pending exception or a derived exception is raised that has a lower priority value than the execution priority being returned to, the PE hardware:

1. Does not unstack the stack.
2. Takes the pending exception or derived exception.
 - The PE will **tail-chain** any pending exception or derived exception on exception return if the pending or derived exception has a lower priority value than the execution priority being returned to.
If the pending or derived exception is escalated to **HardFault** and the execution priority is higher than that of the **HardFault** the PE will enter **Lockup**.
 - The PE will **tail-chain** any synchronous fault on exception return if the synchronous exception has higher priority than the execution priority being returned to.
3. When **tail-chaining** the PE will not execute any instructions from the background state that has been preempted by the exception.

I_{FJWK} *Tail-chaining* is an optimization. It removes unstacking and stacking operations. In the following example the second exception is a *tail-chained exception*:

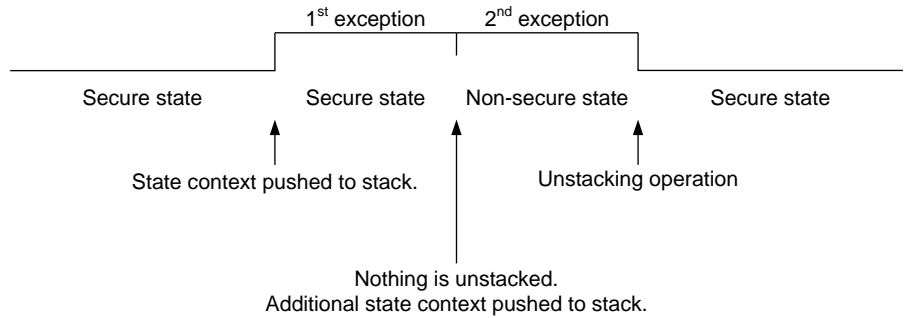
All in Non-secure state:



I_{RWDI} If **tail-chaining** prevents a derived exception on exception return, the derived exception might instead be generated on the return from the last tail-chained exception.

R_{FXVB} When the Background state is Secure state, if **tail-chaining** causes a change of Security state from Secure to Non-secure, additional context is saved on taking the Non-secure exception if it has not already been saved as indicated by **EXC_RETURN.DCRS**:

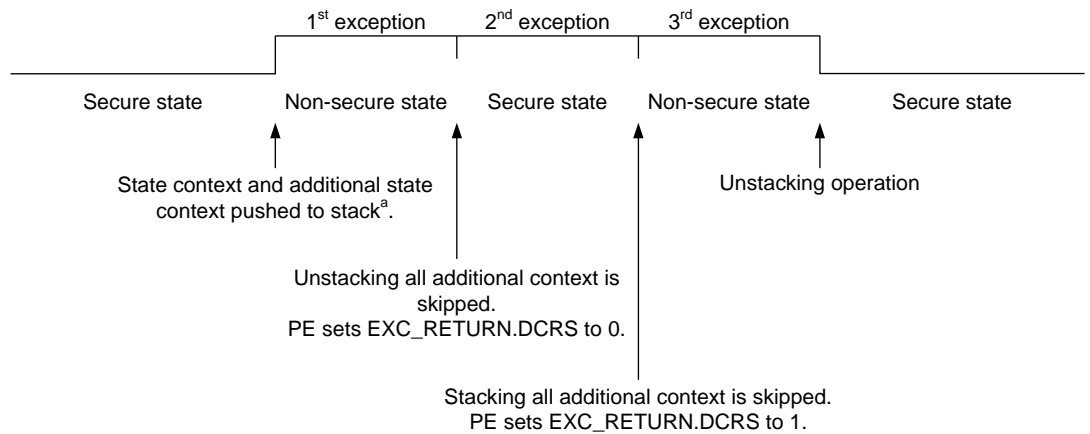
In a PE without the FP Extension:



The extension requirements are - **S**.

I_{TKLM}

When multiple exceptions are **tail-chained**, **EXC_RETURN.DCRS** keeps track of whether the additional context is stacked. The following figure is an example:



a In a PE with the FP Extension, FP context and additional FP context is also stacked if **CONTROL.FPCA** is 1.

I_{TMVF}

When multiple exceptions are **tail-chaining**, a Secure tail-chained exception after a Non-secure exception cannot rely on any registers containing the values they had when no exception was active.

The extension requirements are - **S**.

I_{LNPQ}

Arm recommends that **FPCCR.CLRONRET** is set to 1, to ensure hardware automatically clears the Floating-point context registers to zero on exception return.

The extension requirements are - **FP**.

R_{JMHS}

If the PE recognizes a new asynchronous exception while it is **tail-chaining**, and the new asynchronous exception has a higher priority than the next tailed-chained exception, the PE can, instead, take the new asynchronous exception, using late-arrival preemption.

This rule is true even if the next tail-chained exception is a derived exception on exception return. The PE can, instead, take the new asynchronous exception. If it does, the derived exception becomes pending.

See also:

[B3.19 Exception entry, context stacking on page 89.](#)

[B3.25 Exceptions during exception return on page 107.](#)

B3.27 Exceptions, instruction resume, or instruction restart

- R_{PGRC}** The PE can take an exception during execution of a Load Multiple or Store Multiple instruction, effectively halting the instruction, and resume execution of the instruction after returning from the exception. This is called *instruction resume*.
- The extension requirements are - **M**.
- R_{KRLI}** The PE can abandon execution of a Load Multiple or Store Multiple instruction to take an exception, and after returning from the exception, restart the Load Multiple or Store Multiple instruction again from the start of the instruction. This is called *instruction restart*.
- R_{KCMD}** To support *instruction restart*, singleword load instructions do not update the destination register when the PE takes an exception during execution.
- I_{NDQT}** Instructions that the PE can halt to use instruction resume are called *interrupt-continuable instructions*.
- R_{LGPQ}** The interrupt-continuable instructions are **LDM**, **LDMDB**, **STM**, **STMDB**, **POP (multiple registers)**, and **Push (multiple registers)**.
- The extension requirements are - **M**.
- R_{RDHK}** In a PE with the Floating-point Extension, the floating-point interrupt-continuable instructions are **VLDM**, **VLLDM**, **VLSTM**, **VSTM**, **VPOP**, and **VPUSH**.
- The extension requirements are - **FP**.
- R_{VFBX}** Where a fault is taken during the execution of a **VLLDM** instruction the PE abandons the stacking of the Secure floating-point register contents and save the state so that on return from the fault the instruction can be restarted.
- The extension requirements are - **S && FP**.
- R_{QWWW}** It is IMPLEMENTATION DEFINED whether a **VLLDM** and **VLSTM** or instruction aborts or completes when an interrupt occurs.
- The extension requirements are - **M**.
- R_{QVFC}** When the PE is using instruction resume, **EPSR.ICI** is set to a nonzero value that is the continuation state of the interrupt-continuable instruction:
- For **LDM**, **LDMDB**, **STM**, **STMDB**, **POP (multiple registers)**, and **PUSH (multiple registers)** instructions, **EPSR.ICI** contains the number of the first register in the register list that is to be loaded or stored after instruction resume.
 - For the floating-point instructions **VLDM**, **VSTM**, **VPOP**, and **VPUSH**, **EPSR.ICI** contains the number of the lowest numbered doubleword Floating-point Extension register that was not loaded or stored before the PE took the exception.

The **EPSR.ICI** values shown in the following table are valid **EPSR.ICI** values:

EPSR[26:25]	EPSR[15:12]	EPSR[11:10]
ICI[7:6] = 0b00	ICI[5:2] = reg_num	ICI[1:0] = 0b00
ICI[7:6] = 0b00	ICI[5:2] = 0b0000	ICI[1:0] = 0b00

Note, some instructions listed require **FP**.

R_{XFGN} Behavior is UNPREDICTABLE if **EPSR.ICI** contains a valid **EPSR.ICI** nonzero value and the register number that it contains is either:

- Not in the register list of the interrupt-continuable instructions.
- The first register in the register list of the interrupt-continuable instructions.

The extension requirements are - **M**.

R_{LRGK} The PE generates an INVSTATE UsageFault if **EPSR.ICI** contains a valid nonzero value and the instruction being executed is not an instruction which supports interrupt-continuation. A fault is not generated if the instruction is a **BKPT** instruction.

The extension requirements are - **M**.

R_{JXKQ} If the PE uses instruction resume during a interrupt-continuable instruction, other than a store multiple instruction, then after the exception return, the values of all registers in the register list are UNKNOWN, except for the following:

- Registers that are marked by **EPSR.ICI** as already loaded.
- The base register.
- The **PC**.

The extension requirements are - **M**.

I_{JJQX} If the PE is using instruction restart, Arm recommends that Load Multiple or Store Multiple instructions are not used with data in volatile memory.

R_{NKNQ} When a Load Multiple instruction has the **PC** in its register list, if the PE uses instruction resume or instruction restart during the instruction:

- If the **PC** is loaded before generation of the exception, the PE restores the **PC** before taking the exception, so that after the exception the PE returns to either:
 - Continue execution of the Load Multiple instruction, if the PE used instruction resume.
 - Restart the Load Multiple instruction, if the PE used instruction restart.

Note, Instruction resume requires **M**.

R_{LSCQ} In a PE without the Main Extension, if the PE takes any exception during any Load Multiple or Store Multiple instruction, including **PUSH (multiple registers)** and **POP (multiple registers)**, the PE uses instruction restart and the **Base register** is restored to the original value.

The extension requirements are - **!M**.

R_{RFGF} In a PE with the Main Extension, if the PE takes an exception during any Load Multiple or Store Multiple instruction, including **PUSH (multiple registers)** and **POP (multiple registers)**:

- If the instruction is not in an IT block and the exception is an asynchronous exception, the PE uses instruction resume and **EPSR.ICI** holds the continuation state. The base register is restored to the original value except in the following cases:

Interrupt of an instruction that is using **SP as the base register**

The **SP** that is presented to the exception entry sequence is lower than any element pushed by an **STM**, or not yet popped by an **LDM**.

For Decrement Before (**DB**) variants of the instruction, the **SP** is set to the final value. This is the lowest value in the list.

For Increment After (**IA**) variants of the instruction, the **SP** is restored to the initial value. This is the lowest value in the list.

Interrupt of an instruction that is not using **SP** as the base register

The base register is set to the final value, whether the instruction is a Decrement Before (DB) variant or an Increment After (IA) variant.

- For all other cases:
 - The PE uses instruction restart and the base register is restored to the original value. If the instruction is not in an IT block, **EPSR.ICI** is cleared to zero.

*The extension requirements are - **M**.*

R_{SGWB}

When a Load Multiple instruction includes its **Base register** in its register list, if the PE takes an exception during the instruction:

- The **Base register** is restored to the original value, and:
 - If the instruction is in an IT block, the PE uses instruction restart.
 - If the instruction is not in an IT block, and the PE takes the exception after it loads the **Base register**, **EPSR.IT/ICI** can be set to an IMPLEMENTATION DEFINED value that will load at least the **Base register** and subsequent locations again after returning from the interrupt.

*The extension requirements are - **M**.*

B3.28 Vector tables

R_{NWFF} In a PE with the Security Extension, two vector tables are implemented, the Secure Vector table and the Non-secure Vector table, and it is IMPLEMENTATION DEFINED which of the following is true:

- The PE supports configurability of each vector table base, and two Vector Table Offset Registers, **VTOR_S** and **VTOR_NS**, are provided for this purpose.
- The PE does not support configurability of either vector table base, and **VTOR_S** and **VTOR_NS** are WI.

If the PE supports configurability of each vector table base:

- Exceptions that target Secure state use **VTOR_S** to determine the base address of the Secure vector table.
- Exceptions that target Non-secure state use **VTOR_NS** to determine the base address of the Non-secure vector table.

The extension requirements are - **S**.

R_{GTJQ} In a PE without the Security Extension, a single vector table is implemented, and it is IMPLEMENTATION DEFINED which of the following is true:

- The PE supports configurability of the vector table base, and a single Vector Table Offset Register, **VTOR**, is provided for this purpose.
- The PE does not support configurability of the vector table base, and **VTOR** is WI.

The extension requirements are - **!S**.

I_{WFGX} Arm recommends that **VTOR_S** points to memory that is Secure and not Non-secure callable.

The extension requirements are - **S**.

R_{WPRT} A vector table contains both:

- The initialization value for the main stack pointer on reset.
- The start address of each exception handler.

The *exception number* defines the order of entries.

Word offset in vector table	Value that is held at offset
0	Initial value for the main stack pointer on reset.
1	Start address for the reset handler.
Exception number	Start address for the handler for the exception with that number
.	.
.	.
.	.
Exception number	Start address for the handler for the exception with that number

R_{LFDL} In a PE with a configurable vector table base, the vector table is naturally aligned to a power of two, with an alignment value that is:

- A minimum of 128 bytes.
- Greater than or equal to (Number of Exceptions supported x4).

R_{VDPD} Vector fetches for entries beyond the natural alignment of the associated **VTOR** occur from an UNKNOWN entry within the vector table.

I_{PLSB} Arm recommends that it is ensured that the vector table and **VTOR** are aligned so that the entry for the highest

taken exception falls within the natural alignment of the table, and at a minimum that the vector table is 128 byte aligned. A PE might impose further restrictions on the [VTOR](#).

R_{ZVWS} If a vector fetch causes a Security attribution unit violation or an implementation defined attribution unit violation or a BusFault, a secure VECTTBL HardFault is raised. If the exception priority prevents any secure VECTTBL HardFault preempting, one of the following occurs:

- The PE enters [Lockup](#) at the priority of the original exception.
- The original exception transitions from the pending to the active state.

The extension requirements are - [S](#). Note, SAU and IDAU require [M](#).

R_{XPP} For all vector table entries other than the entry at offset 0, if bit[0] is not set to 1, the first instruction in the exception results in an INVSTATE UsageFault or a HardFault.

I_{BVSC} For all vector table entries other than the entry at offset 0, bit[0] defines [EPSR.T](#) on exception entry. Setting bit[0] to 1 indicates that the exception handler is in the T32 instruction set state.

See also:

[B8.4 IMPLEMENTATION DEFINED Attribution Unit \(IDAU\)](#) on page 222.

[B8.3 Security attribution unit \(SAU\)](#) on page 221.

[B3.9 Exception numbers and exception priority numbers](#) on page 65.

[B3.5.2 Execution Program Status Register \(EPSR\)](#) on page 60.

B3.29 Hardware-controlled priority escalation to HardFault

R_{GNVS} If a synchronous exception with an equal or lower priority value to execution is pending, the PE hardware escalates it to become a HardFault. This rule applies to all synchronous exceptions and DebugMonitor exceptions that are caused by the **BKPT** instruction. This rule does not apply to asynchronous exceptions and all other DebugMonitor exceptions. If the HardFault cannot be taken the PE enters **Lockup**

Note, DebugMonitor exception requires M.

R_{HPLM} **FPCCR.*RDY** (not the current execution priority) determines the escalation of synchronous exceptions generated because of lazy floating-point state preservation. This means that an asynchronous exception might be pended.

R_{PBJQ} When current execution has a priority value less than or equal to the configurable priority exceptions, if a disabled configurable priority exception occurs:

- If it is a synchronous exception, the PE hardware escalates the exception to become a HardFault.
- If it is an asynchronous exception, the PE does not escalate the interrupt. The interrupt remains pending.

R_{DQRR} A fault that has been escalated to a HardFault, and not pended, retains the return address behavior of the original fault and sets **HFSR.FORCED** to 1.

See also:

[B3.9 Exception numbers and exception priority numbers on page 65.](#)

[DebugMonitor exception.](#)

[B3.31 Lockup on page 119.](#)

[B3.11 Security states, exception banking on page 70.](#)

B3.30 Special-purpose mask registers, PRIMASK, BASEPRI, FAULTMASK, for configurable priority boosting

I_BNJJG

In a PE with the Main Extension, the [PRIMASK](#), [FAULTMASK](#), and [BASEPRI](#) registers can be used as follows. A PE without the Main Extension implements [PRIMASK](#), but does not implement [FAULTMASK](#) and [BASEPRI](#).

PRIMASK

In a PE without the Security Extension:

- Setting this bit to one boosts the current execution priority to 0, masking all exceptions with an equal or lower priority.

In a PE with the Security Extension:

- Setting [PRIMASK_S](#) to one boosts the current execution priority to 0.
- If [AIRCR.PRIS](#) is:

0:

Setting [PRIMASK_NS](#) to one boosts the current execution priority to 0.

1:

Setting [PRIMASK_NS](#) to one boosts the current execution priority to 0×80 .

In a PE with the Security Extension, when the current execution priority is boosted to a particular value, all exceptions with an equal or lower priority are masked.

FAULTMASK

In a PE without the Security Extension:

- Setting this bit to one boosts the current execution priority to -1, masking all exceptions with an equal or lower priority.

In a PE with the Security Extension, if [AIRCR.BFHFNMIN](#) is:

0:

Setting [FAULTMASK_S](#) to one boosts the current execution priority to -1.

If [AIRCR.PRIS](#) is:

0: Setting [FAULTMASK_NS](#) to one boosts the current execution priority to 0.

1: Setting [FAULTMASK_NS](#) to one boosts the current execution priority to 0×80 .

1:

Setting [FAULTMASK_S](#) to one boosts the current execution priority to -3.

Setting [FAULTMASK_NS](#) to one boosts the current execution priority to -1.

In a PE with the Security Extension, when the current execution priority is boosted to a particular value, all exceptions with an equal or lower priority are masked.

BASEPRI

In a PE without the Security Extension:

- This field can be set to a priority number between 1 and the maximum supported priority number. This boosts the current execution priority to that number, masking all exceptions with an equal or lower priority.

In a PE with the Security Extension:

- **BASEPRI_S** can be set to a priority number between 1 and the maximum supported priority number.
- If **AIRCR.PRIS** is:
 - 0**: **BASEPRI_NS** can be set to a priority number between 1 and the maximum supported priority number.
 - 1**: **BASEPRI_NS** can be set to a priority number between 1 and the maximum supported priority number. The value in **BASEPRI_NS** is then mapped to the bottom half of the priority range, so that the current execution priority is boosted to the mapped-to value in the bottom half of the priority range, that is from 0×80 to the supported maximum.

In a PE with the Security Extension, when the current execution priority is boosted to a particular value, all exceptions with an equal or lower priority are masked.

Note, **FAULTMASK** and **BASEPRI** require *M*.

R_FHMC The **PRIMASK**, **FAULTMASK**, and **BASEPRI** priority boosting mechanisms only boost the group priority, not the subpriority.

Note, **FAULTMASK** and **BASEPRI** require *M*.

R_SKB_J Without the Security Extension:

- An exception return sets **FAULTMASK** to 0.

The extension requirements are - *!S* && *M*.

R_HRTM With the Security Extension:

- An exception return sets **FAULTMASK** to 0 if the *raw execution priority* is greater than or equal to 0. **EXC_RETURN.ES** indicates which banked instance of **FAULTMASK** is set to 0.

The extension requirements are - *S* && *M*.

I_LSX_J The *raw execution priority* is:

- The execution priority minus the effects of any configurable **PRIMASK**, **FAULTMASK**, or **BASEPRI** priority boosting.

The extension requirements are - *M*.

I_GBV_L The *requested execution priority* for a Security state is negative when any of the following are true:

- The banked **FAULTMASK** bit is 1, including when **AIRCR.PRIS** is also 1.
- A HardFault is active.
- An NMI is active and targets the Security state for which the requested execution priority is being calculated .

Note, **FAULTMASK** requires *M*.

See also:

[B3.13 Priority model on page 77.](#)

[B3.9 Exception numbers and exception priority numbers on page 65.](#)

B3.31 Lockup

I_{RKJB} *Lockup* is a PE state where the PE stops executing instructions in response to an error for which escalation to an appropriate HardFault handler is not possible because of the current execution priority. An example is a synchronous exception that escalates to a Secure HardFault, but cannot escalate to a Secure HardFault because a Secure HardFault is already active.

I_{FSFR} Arm recommends that an implementation provides a **LOCKUP** signal that, when the PE is in *lockup*, signals to the external system that the PE is in *lockup*.

R_{MBTM} When the PE is in *lockup*:

- **DHCSR.S_LOCKUP** reads as 1.
- The **PC** reads as 0xFFFFFFFF. This is an XN address.
- The PE stops fetching and executing instructions.
- If the implementation provides an external **LOCKUP** signal, **LOCKUP** is asserted HIGH.

R_{JRJC} Exit from *lockup* is only by one of the following:

- A Cold reset.
- A Warm reset.
- Entry to Debug state.
- Preemption by another exception.

Note, entry to Debug state requires Halting debug.

R_{HJNP} Exit from *lockup* causes both **DHCSR.S_LOCKUP** and, if implemented, the external **LOCKUP** signal, to be deasserted.

R_{SPPN} On an exit from *lockup* by entry to Debug state, or by preemption by another exception, the return address is 0xFFFFFFFF.

Note, entry to Debug state requires Halting debug.

I_{CRHJ} After exit from *lockup* by entry to Debug state, or by preemption by another exception, a subsequent return from Debug state or that exception without modifying the return address attempts to execute from 0xFFFFFFFF. Execution from this address is guaranteed to generate an IACCVIOL MemManage fault, causing the PE to reenter *lockup* if the execution priority has not been modified. Modification of the return address would enable execution to be resumed, however Arm recommends treating entry to *lockup* as fatal and requiring the PE to be reset.

Note, entry to Debug state requires Halting debug.

See also:

[B3.13 Priority model on page 77.](#)

[Chapter B11 Debug on page 230.](#)

B3.31.1 Instruction-related lockup behavior

Instruction execution

R_{VGMR}A synchronous exception results in **lockup** when:

- The synchronous exception would otherwise escalate to a Secure HardFault and any of the following is true:
 - Secure HardFault is already active.
 - NMI is active and **AIRCR.BFHFNMINs** is 0.
 - **FAULTMASK_S.FM** is 1.
 - Non-secure HardFault is active and **AIRCR.BFHFNMINs** is 0.
- The synchronous exception would otherwise escalate to a Non-secure HardFault and any of the following is true:
 - Non-secure HardFault or Secure HardFault is already active.
 - NMI is active.
 - **FAULTMASK_NS.FM** or **FAULTMASK_S.FM** is 1.

*The extension requirements are - **S**. Note, **FAULTMASK** requires **M**.***R_{QMMB}**If the Security Extension is not implemented, a synchronous exception results in **lockup** when:

- The synchronous exception would otherwise escalate to HardFault and any of the following is true:
 - HardFault is already active.
 - NMI is active.
 - **FAULTMASK** is always 1.

*The extension requirements are - **!S**.***R_{VGNW}**Entry to **lockup** from an exception causes:

- Any Fault Status Registers associated with the exception to be updated.
- No update to the pending exception state or to the active exception state.
- The **PC** to be set to 0xEFFFFFFE.
- **EPSR.IT** to become UNKNOWN.

In addition, **HFSR.FORCED** is not changed.**R_{DWKEP}**Asynchronous BusFaults do not cause **lockup**.**R_{KTQM}**When a BusFault does not cause **lockup**, the value that is read or written to the location that generated the BusFault is UNKNOWN.**R_{HTVD}**ITSTATE does not advance when the PE is in **lockup**.

Floating-point lazy Floating-point context preservation

R_{RNKB}When **FPCCR.LSPACT** is 1, a NOCP UsageFault, **AU** violation, **MPU** violation, or synchronous BusFault during lazy Floating-point context preservation causes **lockup** if any of the following is true:

- **FPCCR.HFRDY** is 0, the *RDY bit associated with the original exception is 0, and the current execution priority is high enough to prevent preemption.

*The extension requirements are - **FP**. Note, an MPU violation requires MPU, an SAU violation requires S.***R_{MMBJ}**When **FPCCR.LSPEN** is 0, any faults that are caused by floating-point register reads or writes during exception entry or exception return are handled as faults on stacking or unstacking respectively.

The extension requirements are - *FP*.

B3.31.2 Exception-related lockup behavior

Vector or stack pointer error on reset

R_{BHVG}

On reset, if reading the vector table to obtain either the vector for the reset handler or the initialization value for the main stack pointer causes a BusFault, the PE enters **lockup** in HardFault with the following behavior:

- **HFSR.VECTTBL** is set to 1.
- In a PE with the Security Extension, Secure HardFault is made active. That is, **SHCSR.S.HARDFFAULTACT** is set to 1.
- In a PE without the Security Extension, HardFault is made active. That is, **SHCSR.HARDFFAULTACT** is set to 1.
- An UNKNOWN value is loaded into the main stack pointer.
- The **IPSR** is set to 0.
- **EPSR.T** is UNKNOWN.
- **EPSR.IT** is set to zero.
- The **PC** is set to 0xEFFFFFFE.

Note, a Secure HardFault requires *S*.

Errors on preemption and stacking for exception entry

R_{VKTX}

An **AU** violation, **MPU** violation, NOCP UsageFault, STKOF UsageFault, LSERR SecureFault, or synchronous BusFault during context stacking causes **lockup** when:

- The exception would escalate to a Secure HardFault if any of the following is true:
 - Secure HardFault is already active.
 - NMI is active and **AIRCR.BFHFNMINs** is 0.
 - **FAULTMASK_S.FM** is 1.
 - Non-secure HardFault is active and **AIRCR.BFHFNMINs** is 0.
- The exception would escalate to a Non-secure HardFault if any of the following is true:
 - Non-secure HardFault or Secure HardFault is already active.
 - NMI is active.
 - **FAULTMASK_NS.FM** or **FAULTMASK_S.FM** is 1.

In these cases, the point of PE **lockup** is when, after the exception to be taken has been chosen, the handler for that exception is entered. These cases do not in themselves cause any additional exception to become pending.

The extension requirements are - *S*. Note, an AU violation requires *S*, an MPU violation requires *MPU*, a UsageFault requires *M*, a SecureFault requires *S*.

R_{QSSB}

When an **AU** violation, **MPU** violation, NOCP UsageFault, STKOF UsageFault, LSERR SecureFault, or synchronous BusFault occurs during context stacking, it is IMPLEMENTATION DEFINED whether the PE continues to stack any of the remaining context.

The extension requirements are - *S*. Note, an AU violation requires *S*, an MPU violation requires *MPU*, a UsageFault requires *M*, a SecureFault requires *S*, LSERR requires *FP*.

R_{GJJG}

At the point of encountering an **AU** violation, **MPU** violation, NOCP UsageFault, STKOF UsageFault, LSERR SecureFault, or synchronous bus error during context stacking, the PE:

- Updates any Fault Status Registers associated with the error.
- Does not change **HFSR.FORCED**.

At the point of **lockup**:

- All state, including the **LR**, **IPSR**, and active and pending bits, is modified as though the fault on context stacking had never occurred, other than the following:
 - **EPSR.T** becomes UNKNOWN.
 - **EPSR.IT** is set to zero.
 - The **PC** is set to 0xEFFFFFFE.

The extension requirements are - **S**. Note, an **AU** violation requires **S**, an **MPU** violation requires **MPU**, a **UsageFault** requires **M**, a **SecureFault** requires **S**, **LSERR** requires **FP**.

Vector read error on NMI or HardFault entry

R_{CTKP}

On entry to an NMI or HardFault, if reading the vector table to obtain the vector for the NMI or HardFault handler causes a bus error, the PE enters **lockup** with the following behavior:

- **HFSR.VECTTBL** is set to 1.
- The **IPSR** is updated to hold the exception number of the exception taken.
- The active bit of the exception that is taken is set to 1.
- The pending bit of the exception that is taken is cleared to 0.
- **EPSR.T** is UNKNOWN.
- **EPSR.IT** is set to zero.
- The **LR** is set to the **EXC_RETURN** value that would have been used had the fault not occurred.
- The **PC** is set to 0xEFFFFFFE.

I_{NMRW}

Because **AU** violations on vector reads are required to be treated as late-arriving, they cannot cause **lockup**, and instead result in a higher priority exception being taken. Vector reads always use the default memory map and cannot generate **MPU** violations.

Integrity checks on exception return

R_{TRFJ}

A fault that is generated by a failed *integrity check on exception return* is generated after either the active bit for the returning exception, or the active bit for NMI or HardFault, has been cleared to 0, and if applicable, after **FAULTMASK** has also been cleared to 0. A fault that is generated by a failed integrity check on exception return causes **lockup** when:

- The exception would escalate to a Secure HardFault and any of the following is true:
 - Secure HardFault is already active.
 - NMI is active and **AIRCR.BFHFNMINs** is 0.
 - **FAULTMASK_S.FM** is 1.
 - Non-secure HardFault is active and **AIRCR.BFHFNMINs** is 0.
- The exception would escalate to a Non-secure HardFault and any of the following is true:
 - Non-secure HardFault or Secure HardFault is already active.
 - NMI is active.
 - **FAULTMASK_NS.FM** or **FAULTMASK_S.FM** is 1.

The extension requirements are - **S**.

R_{DFKP}

When the PE enters **lockup** because of a fault that is generated by a failed integrity check, the PE:

- Updates any Fault Status Registers associated with the error.

- Sets `IPSR` to 0, if `EXC_RETURN` for the returning exception indicated a return to Thread mode.
- Sets `IPSR` to 3, if `EXC_RETURN` for the returning exception indicated a return to Handler mode.
- Sets the stack pointer that is used for unstacking to the value it would have had if the fault had not occurred.
 - If the `XPSR` load faults, the `SP` is 64-bit aligned.
- Updates `CONTROL.FPCA`, based on `EXC_RETURN.FType`.
- `CONTROL.SFPA` becomes UNKNOWN.
- Sets the `PC` to `0xEFFFFFFE`.

In addition, the `APSR`, `EPSR`, `FPSCR`, `R0-R12`, `LR`, and `S0-S31` are UNKNOWN.

Note, CONTROL.FPCA and SFPA, FPSCR and S0-S31 require FP.

Errors when unstacking state on exception return

R_{WKSJ}

Context unstacking is performed after any clearing of exception active bits or `FAULTMASK`, that is required by the exception return, has been made visible. A synchronous exception during context unstacking causes **lockup** when:

- The exception would escalate to a Secure HardFault and any of the following is true:
 - Secure HardFault is already active.
 - `FAULTMASK_S.FM` is 1.
 - Non-secure HardFault is active and `AIRCR.BFHFNMINS` is 0.
- The exception would escalate to a Non-secure HardFault and any of the following is true:
 - Non-secure HardFault or Secure HardFault is already active.
 - NMI is active.
 - `FAULTMASK_NS.FM` or `FAULTMASK_S.FM` is 1.

The extension requirements are - S.

R_{XFCQ}

When a synchronous exception during context unstacking causes **lockup**, the PE:

- Updates any Fault Status Registers associated with the error.
- Sets `IPSR` to 0, if `EXC_RETURN` for the returning exception indicated a return to Thread mode.
- Sets `IPSR` to 3, if `EXC_RETURN` for the returning exception indicated a return to Handler mode.
- Sets the stack pointer that is used for unstacking to the value it would have had if the fault had not occurred.
 - If the `XPSR` load faults, the `SP` is 64-bit aligned.
- Updates `CONTROL.FPCA`, based on `EXC_RETURN`.
- Sets the `PC` to `0xEFFFFFFE`.

In addition, the `APSR`, `EPSR`, `FPSCR`, `R0-R12`, `LR`, and `S0-S31` are UNKNOWN.

See also:

[B3.22 Exception return on page 101.](#)

B3.32 Context Synchronization Event

- R_{QXWD}** The architecture requires a Context synchronization event to guarantee visibility of any change to any memory-mapped register described in the architecture. Following a [Content synchronization event](#) a completed write to a memory-mapped register is visible to an indirect read by an instruction appearing in program order after the context synchronization event.
- R_{TVHX}** Between any change to a memory-mapped register and a subsequent [Content synchronization event](#), it is UNPREDICTABLE whether an indirect read of the register by the PE uses the old or new values.
- R_{RMM}** Where multiple changes are made to memory-mapped registers before a [Content synchronization event](#), each value might independently be the old or new value.
- R_{NSLQ}** Where unsynchronized values apply to different areas of architectural functionality, or IMPLEMENTATION DEFINED functionality, those areas might independently treat the values as being either the old or new value.
- R_{BKSX}** The choice between the behaviors is IMPLEMENTATION DEFINED and might vary for each use of the unsynchronized value.

B3.33 Coprocessor support

- R_{BSLX}** Coprocessor support is OPTIONAL.
*The extension requirements are - **M**.*
- I_{JBMG}** When coprocessors are not supported, the fields in **CPACR**, **NSACR**, and **CPPWR** that are associated with the unsupported coprocessor are RAZ/WI.
*The extension requirements are - **M**.*
- R_{XSQH}** The architecture supports 0-16 coprocessors, CP0 to CP15.
*The extension requirements are - **M**.*
- R_{HJDH}** CP0 to CP7 are IMPLEMENTATION DEFINED.
*The extension requirements are - **M**.*
- R_{XPRQ}** It is IMPLEMENTATION DEFINED whether CP0 to CP7 can be used from both Secure and Non-secure states or whether the coprocessor is enabled for only Secure or Non-secure state.
*The extension requirements are - **M**. Note, Secure state requires S.*
- R_{QSRC}** Arm reserves CP8 to CP15.
*The extension requirements are - **M**.*
- R_{LKZM}** CP10 to CP11 are reserved to support the Floating-point Extension, and CP10 controls the CP11 Floating-point instructions.
*The extension requirements are - **M**.*
- R_{LPMK}** The state that is associated with Floating-point unit described in **CPPWR.SU10** applies to S registers, D registers, and FPSCR.
*The extension requirements are - **FP**.*
- R_{XXDG}** Instructions that are issued to unimplemented or disabled coprocessors result in a NOCP UsageFault.
*The extension requirements are - **M**.*
- R_{RMLV}** If a coprocessor cannot complete an instruction, an UNDEFINSTR UsageFault is generated.
*The extension requirements are - **M**.*

See also:

[Chapter B4 Floating-point Support on page 126.](#)

[CPACR, Coprocessor Access Control Register](#)

[CPPWR, Coprocessor Power Control Register](#)

Chapter B4

Floating-point Support

This chapter specifies the Armv8-M Floating-point support rules. It contains the following sections:

- [B4.1 *The optional Floating-point Extension, Fpv5* on page 127.](#)
- [B4.2 *About the Floating-point Status and Control Registers* on page 129.](#)
- [B4.3 *Registers for Floating-point data processing, S0-S31, or D0-D15* on page 130.](#)
- [B4.4 *Floating-point standards and terminology* on page 131.](#)
- [B4.5 *Floating-point data representable* on page 132.](#)
- [B4.6 *Floating-point encoding formats, half-precision, single-precision, and double-precision* on page 133.](#)
- [B4.7 *The IEEE 754 Floating-point exceptions* on page 135.](#)
- [B4.8 *The Flush-to-zero mode* on page 136.](#)
- [B4.9 *The Default NaN mode, and NaN handling* on page 138.](#)
- [B4.10 *The Default NaN* on page 139.](#)
- [B4.11 *Combinations of Floating-point exceptions* on page 140.](#)
- [B4.12 *Priority of Floating-point exceptions relative to other Floating-point exceptions* on page 141.](#)

B4.1 The optional Floating-point Extension, Fpv5

I_VBNH The optional Floating-point Extension defines a *Floating Point Unit* (FPU). Coprocessors 10 and 11 support the Extension.

The extension requirements are - FP.

I_RXQX Floating-point is sometimes abbreviated to FP.

The extension requirements are - FP.

R_QOBM The version of Floating-point Extension that is supported is Fpv5.

The extension requirements are - FP.

I_FGSG Fpv5 provides all of the following:

- Single-precision arithmetic operations.
- Optional double-precision arithmetic operations.
- Conversions between integer, double-precision, single-precision, and half-precision formats.
- Registers for Floating-point processing S0-S31, or D0-D15.
- Data transfers, between Arm general-purpose registers and Fpv5 Extension registers S0-S31, or D0-D15, of single-precision and double-precision values.
- A [Flush-to-zero mode](#) that software can enable or disable.
- An optional *alternative half-precision* interpretation of the IEEE 754 half-precision encoding format.

Fpv5 adds the following System registers:

- The [FPSCR](#), to the CP10 and CP11 System register space.
- The [FPCAR](#), [FPCCR](#), [FPDSCR](#), [MVFR0](#), [MVFR1](#), and [MVFR2](#), to the *System Control Block* (SCB).

The extension requirements are - FP.

I_PVBQ When the Floating-point Extension is implemented, some software tools might require the following information:

Extension	Single-precision arithmetic operations only	Single and double-precision arithmetic operations
Fpv5	Fpv5-SP-D16-M	Fpv5-D16-M

The extension requirements are - FP.

I_FTDS When the Floating-point Extension is implemented, software can interrogate [MVFR0](#), [MVFR1](#), and [MVFR2](#) to discover the Floating-point features that are implemented.

The extension requirements are - FP.

I_JDJQ To use the Floating-point Extension, software must enable access to CP10, by writing to [CPACR.CP10](#).

The extension requirements are - FP.

R_PDMV The value of [CPACR.CP11](#) is UNKNOWN if it is not programmed to the same value as [CPACR.CP10](#).

The extension requirements are - FP.

See also:

[B6.1 System address map on page 199.](#)

[B4.2 About the Floating-point Status and Control Registers on page 129.](#)

[B4.3 Registers for Floating-point data processing, S0-S31, or D0-D15 on page 130.](#)

[B4.8 The Flush-to-zero mode on page 136.](#)

Chapter B4. Floating-point Support

B4.1. The optional Floating-point Extension, Fpv5

[B4.9 *The Default NaN mode, and NaN handling* on page 138.](#)

[B4.6 *Floating-point encoding formats, half-precision, single-precision, and double-precision* on page 133.](#)

B4.2 About the Floating-point Status and Control Registers

R_{HCJS} The register map of the coprocessor System register space is as follows.

Location	Register	Information
0b0001	FPSCR.{N,Z,C,V}	Access to flags

All locations that are not explicitly listed in this table are reserved, and accesses to these locations result in UNPREDICTABLE behavior.

*The extension requirements are - **FP**.*

I_{GJWP} Software can use [VMRS](#) and [VMSR](#) instructions to access the Floating-point Status and Control registers.

*The extension requirements are - **FP**.*

R_{FXBJ} Execution of Floating-point instructions that generate Floating-point exceptions update the appropriate status fields of [FPSCR](#).

*The extension requirements are - **FP**.*

See also:

[B3.33 Coprocessor support](#) on page 125.

[B4.1 The optional Floating-point Extension, Fpv5](#) on page 127.

[FPSCR, Floating Point Status and Control Register](#).

B4.3 Registers for Floating-point data processing, S0-S31, or D0-D15

R_{TWCB}

The registers that FPv5 adds for Floating-point processing are visible as either:

- 32 single-precision registers, S0-S31.
- 16 double-precision registers, D0-D15.

These map as follows:

S0-S31		D0-D15	
S0	-----	D0	-----
S1		D1	
S2		D2	
S3		D3	
S4			
S5			
S6			
S7			
⋈		⋈	
S28	-----	D14	-----
S29		D15	
S30			
S31			

The extension requirements are - **FP**.

R_{XWJQ}

After a Warm reset, the values of S0-S31 or D0-D15 are UNKNOWN.

The extension requirements are - **FP**.

See also:

[B4.1 The optional Floating-point Extension, FPv5 on page 127.](#)

[B3.18 Exception handling on page 87.](#)

B4.4 Floating-point standards and terminology

\mathbb{I}_{XNMN} There are two editions of the IEEE 754 standard:

- IEEE 754-1985.
- IEEE 754-2008.

In this manual, references to IEEE 754 that do not include the year apply to either edition.

*The extension requirements are - **FP**.*

\mathbb{I}_{MQFS} The Floating-point terminology that this manual uses differs from that used in IEEE 754-2008 as follows:

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

*The extension requirements are - **FP**.*

\mathbb{I}_{BGNP} The following is called *Arm standard Floating-point operation*:

- IEEE 754-2008 plus the following configuration:
 - **Flush-to-zero mode** enabled.
 - Default **NaN** mode enabled.
 - Round to Nearest mode selected.
 - Alternative half-precision interpretation not selected.

*The extension requirements are - **FP**.*

See also:

IEEE 754-2008, IEEE Standard for Floating-point Arithmetic, August 2008.

[B4.8 The Flush-to-zero mode on page 136.](#)

[B4.9 The Default NaN mode, and NaN handling on page 138.](#)

[B4.6 Floating-point encoding formats, half-precision, single-precision, and double-precision on page 133.](#)

B4.5 Floating-point data representable

R_{FWXC} FPv5 supports the following, as defined by IEEE 754:

- Normalized numbers.
- Denormalized numbers.
- Zeros, +0 and -0.
- Infinities, $+\infty$ and $-\infty$.
- NaNs, signaling NaNs and quiet NaN.

*The extension requirements are - **FP**.*

See also:

[B4.4 Floating-point standards and terminology on page 131.](#)

IEEE 754-2008, IEEE Standard for Floating-point Arithmetic, August 2008.

[B4.6 Floating-point encoding formats, half-precision, single-precision, and double-precision on page 133.](#)

The interpretation that is defined by IEEE 754-2008. See the following table:

E (biased exponent)	T (trailing significand)	S (sign bit)	T [51]	Value
Zero for all formats.	Nonzero	-	-	A denormalized number.
-	Zero	0	-	Zero, +0
-	-	1	-	Zero, -0
Zero < E < 0x1F, if one of the half precision formats.	-	-	-	A normalized number.
Zero < E < 0xFF, if single-precision format.	-	-	-	-
Zero < E < 0x7FF, if double-precision format.	-	-	-	-
0x1F, if half-precision format, IEEE interpretation.	Nonzero	-	0	A signaling NaN
0xFF, if single-precision format.	-	-	1	A quiet NaN
0x7FF, if double-precision format.	Zero	0	-	Infinity, +∞
-	Zero	1	-	Infinity, -∞
0x1F, if half-precision, alternative half-precision interpretation.	-	-	-	A normalized number.

The extension requirements are - **FP**.

R_{DPHH}

The value of a normalized number is equal to:

Half-precision: $(-1)^S \times 2^{(E-15)} \times (1.T)$

Single-precision: $(-1)^S \times 2^{(E-127)} \times (1.T)$

Double-precision: $(-1)^S \times 2^{(E-1023)} \times (1.T)$

The value of a denormalized number is equal to:

Half-precision: $(-1)^S \times 2^{-14} \times (0.T)$

Single-precision: $(-1)^S \times 2^{-126} \times (0.T)$

Double-precision: $(-1)^S \times 2^{-1022} \times (0.T)$

The extension requirements are - **FP**.

R_{PKXD}

Denormalized numbers can be flushed to zero. Fpv5 provides a [Flush-to-zero mode](#).

The extension requirements are - **FP**.

See also:

IEEE 754-2008, IEEE Standard for Floating-point Arithmetic, August 2008.

[B4.5 Floating-point data representable on page 132.](#)

B4.7 The IEEE 754 Floating-point exceptions

R_{BCC}

The IEEE 754 Floating-point exceptions are:

Invalid Operation: This exception is as IEEE 754-2008 (7.2) describes.

Division by zero: This exception is as IEEE 754-2008 (7.3) describes, with the following assumption:

- For the reciprocal and reciprocal square root estimate functions the dividend is assumed to be +1.0.

Overflow: This exception is as IEEE 754-2008 (7.4) describes.

Underflow: This exception is as IEEE 754-2008 (7.5) describes, with the additional clarification that:

- Assessing whether a result is tiny and nonzero is done before rounding.

Inexact: This exception is as IEEE 754-2008 (7.6) describes.

*The extension requirements are - **FP**.*

I_{JCWS}

The criteria for the Underflow exception to be generated are different in [Flush-to-zero mode](#).

*The extension requirements are - **FP**.*

I_{NFHK}

The corresponding status flags for the IEEE 754 Floating-point exceptions are **FPSCR**.{**IOC**, **DZC**, **OFC**, **UFC**, **IXC**}.

*The extension requirements are - **FP**.*

See also:

IEEE 754-2008, IEEE Standard for Floating-point Arithmetic, August 2008.

[B4.8 The Flush-to-zero mode on page 136.](#)

B4.8 The Flush-to-zero mode

- I_{XGFP}** Software can enable [Flush-to-zero mode](#) by setting `FPSCR.FZ` to 1.
*The extension requirements are - **FP**.*
- I_{WMKJ}** Using [Flush-to-zero mode](#) is a deviation from IEEE 754.
*The extension requirements are - **FP**.*
- R_{JQHX}** Half-precision Floating-point numbers are exempt from [Flush-to-zero mode](#).
*The extension requirements are - **FP**.*
- R_{VJSF}** When [Flush-to-zero mode](#) is enabled, all single-precision denormalized inputs and double-precision denormalized inputs to Floating-point operations are treated as though they are zero, that is they are flushed to zero.
*The extension requirements are - **FP**.*
- R_{KBJJ}** When an input to a Floating-point operation is flushed to zero, the PE generates an Input Denormal exception.
*The extension requirements are - **FP**.*
- R_{SBCK}** Input Denormal exceptions are only generated in [Flush-to-zero mode](#).
*The extension requirements are - **FP**.*
- R_{WJDM}** When [Flush-to-zero mode](#) is enabled, the sequence of events for an input to a Floating-point operation is:
1. Flush to Zero processing takes place. If appropriate, the input is flushed to zero and the PE generates an Input Denormal exception.
 2. Tests for the generation of any other Floating-point exceptions are done after Flush to Zero processing.
- The extension requirements are - **FP**.*
- R_{PHPT}** When [Flush-to-zero mode](#) is enabled, the result of a Floating-point operation is treated as if it is zero if, before rounding, it satisfies the condition:
 $0 < \text{Abs}(\text{result}) < \text{MinNorm}$, where:
- MinNorm is 2^{-126} for single-precision.
 - MinNorm is 2^{-1022} for double-precision.
- The result is said to be flushed to zero.
*The extension requirements are - **FP**.*
- R_{QPQF}** When the result of a Floating-point operation is flushed to zero, the PE generates an Underflow exception.
*The extension requirements are - **FP**.*
- R_{TPVD}** In [Flush-to-zero mode](#), the PE generates Underflow exceptions only when a result is flushed to zero. This uses different criteria than when [Flush-to-zero mode](#) is disabled.
*The extension requirements are - **FP**.*
- R_{RTPH}** When a Floating-point number is flushed to zero, the sign is preserved. That is, the sign bit of the zero matches the sign bit of the number being flushed to zero.
*The extension requirements are - **M** && **FP**.*
- R_{RWRT}** The PE does not generate an Inexact exception when a Floating-point number is flushed to zero.
*The extension requirements are - **FP**.*

I_{SQCJ} The corresponding status flag for the Input Denormal exception is [FPSCR.IDC](#).

The extension requirements are - [FP](#).

See also:

[B4.7 The IEEE 754 Floating-point exceptions](#) on page 135.

B4.9 The Default NaN mode, and NaN handling

I_{FGPN} Software can enable Default NaN mode by setting **FPSCR.DN** to 1.
The extension requirements are - FP.

I_{DJVH} Using Default NaN mode is a deviation from IEEE 754.
The extension requirements are - FP.

R_{QMQC} When Default NaN mode is enabled, the *Default NaN* is the result of both:

- All Floating-point operations that produce an untrapped Invalid Operation exception.
- All Floating-point operations whose inputs include at least one quiet NaN but no signaling NaNs.

The extension requirements are - FP.

R_{NPRL} IEEE 754 specifies that:

- An operation that produces an untrapped Invalid Operation exception returns a quiet NaN as its result.

When Default NaN mode is disabled, behavior complies with this and adds:

- If the Invalid Operation exception was generated because one of the inputs to the operation was a signaling NaN, the quiet NaN result is equal to the first signaling NaN input with its most significant bit set to 1.
- The quiet NaN result is the Default NaN otherwise.

The *first signaling NaN input* means the first argument, in the left-to-right ordering of arguments, that is passed to the pseudocode function describing the operation.

The extension requirements are - FP.

R_{VCSB} IEEE 754 specifies that:

- An operation using a Quiet NaN as an input, but no signaling NaNs as inputs, returns one of its quiet NaN inputs as its result.

When Default NaN mode is disabled, behavior complies with this and adds:

- The Quiet NaN result is the first Quiet NaN input.

The *first quiet NaN input* means the first argument, in the left-to-right ordering of arguments, that is passed to the pseudocode function describing the operation.

The extension requirements are - FP.

I_{LXLF} Depending on the Floating-point operation, the exact value of a Quiet NaN result might differ in both sign and the number of T bits from its source.
The extension requirements are - FP.

See also:

[B4.10 The Default NaN on page 139.](#)

[B4.6 Floating-point encoding formats, half-precision, single-precision, and double-precision on page 133.](#)

B4.10 The Default NaN

R_FQFG

The Default NaN is:

Field	Half-precision, IEEE 754-2008 interpretation	Single-precision	Double-precision
S	0	0	0
E	0x1F	0xFF	0x7FF
T	bit[9] == 1, bits[8:0] == 0	bit[22] == 1, bits[21:0] == 0	bit[51] == 1, bits[50:0] == 0

The extension requirements are - *FP*.

See also:

[B4.6 Floating-point encoding formats, half-precision, single-precision, and double-precision](#) on page 133.

[B4.9 The Default NaN mode, and NaN handling](#) on page 138.

B4.11 Combinations of Floating-point exceptions

I_{BTT}

In compliance with IEEE 754:

- An Inexact Floating-point exception can occur with an Overflow Floating-point exception.
- An Inexact Floating-point exception can occur with an Underflow Floating-point exception.

*The extension requirements are - **FP**.*

R_{LFVH}

An Input Denormal exception can occur with other Floating-point exceptions.

*The extension requirements are - **FP**.*

See also:

[B4.7 The IEEE 754 Floating-point exceptions on page 135.](#)

[B4.8 The Flush-to-zero mode on page 136.](#)

B4.12 Priority of Floating-point exceptions relative to other Floating-point exceptions

R_{PLHJ}

Some Floating-point instructions specify more than one Floating-point operation. In these cases, an exception on one operation is higher priority than an exception on another operation when generation of the second exception depends on the result of the first operation. Otherwise, it is UNPREDICTABLE which exception is higher priority.

*The extension requirements are - **FP**.*

See also:

[B4.7 The IEEE 754 Floating-point exceptions on page 135.](#)

Chapter B5

Memory Model

This chapter specifies the Armv8-M memory model architecture rules. It contains the following sections:

- [B5.1 *Memory accesses* on page 144.](#)
- [B5.2 *Address space* on page 145.](#)
- [B5.3 *Endianness* on page 146.](#)
- [B5.4 *Alignment behavior* on page 148.](#)
- [B5.5 *Atomicity* on page 149.](#)
- [B5.6 *Concurrent modification and execution of instructions* on page 151.](#)
- [B5.7 *Access rights* on page 153.](#)
- [B5.8 *Observability of memory accesses* on page 155.](#)
- [B5.9 *Completion of memory accesses* on page 157.](#)
- [B5.10 *Ordering requirements for memory accesses* on page 158.](#)
- [B5.11 *Ordering of implicit memory accesses* on page 159.](#)
- [B5.12 *Ordering of explicit memory accesses* on page 160.](#)
- [B5.13 *Memory barriers* on page 161.](#)
- [B5.14 *Normal memory* on page 166.](#)
- [B5.15 *Cacheability attributes* on page 168.](#)
- [B5.16 *Device memory* on page 169.](#)
- [B5.17 *Device memory attributes* on page 171.](#)

- B5.18 *Shareability domains* on page 174.
- B5.19 *Shareability attributes* on page 176.
- B5.20 *Memory access restrictions* on page 177.
- B5.21 *Mismatched memory attributes* on page 178.
- B5.22 *Load-Exclusive and Store-Exclusive accesses to Normal memory* on page 180.
- B5.23 *Load-Acquire and Store-Release accesses to memory* on page 181.
- B5.24 *Caches* on page 183.
- B5.25 *Cache identification* on page 185.
- B5.26 *Cache visibility* on page 186.
- B5.27 *Cache coherency* on page 187.
- B5.28 *Cache enabling and disabling* on page 188.
- B5.29 *Cache behavior at reset* on page 189.
- B5.30 *Behavior of Preload Data (PLD) and Preload Instruction (PLI) instructions with caches* on page 190.
- B5.31 *Branch predictors* on page 191.
- B5.32 *Cache maintenance operations* on page 192.
- B5.33 *Ordering of cache maintenance operations* on page 196.
- B5.34 *Branch predictor maintenance operations* on page 197.

B5.1 Memory accesses

I_{XRDS} The memory accesses that are referred to in describing the memory model are instruction fetches from memory and load or store data accesses.

R_{LKQN} The instruction operation uses the [MemA \(\)](#) or [MemU \(\)](#) helper functions. If the Main Extension is not implemented unaligned accesses using the [MemU \(\)](#) helper functions generate an alignment fault.

R_{BFNF} A memory access is governed by:

- Whether the access is a read or a write.
- The address alignment.
- Data endianness.
- Memory attributes.

See also:

[B5.11 Ordering of implicit memory accesses](#) on page 159.

[B5.12 Ordering of explicit memory accesses](#) on page 160.

[B5.14 Normal memory](#) on page 166.

[B5.16 Device memory](#) on page 169.

[B5.20 Memory access restrictions](#) on page 177.

[B6.2 The System region of the system address map](#) on page 200.

B5.2 Address space

- R_{FFMK}** The address space is a single, flat address space of 2^{32} bytes.
- R_{SNPV}** In the address space, byte addresses are unsigned numbers in the range $0-2^{32}-1$.
- R_{RGBT}** If an address calculation overflows or underflows the address space, it wraps around. Address calculations are modulo 2^{32} .
- I_{JTKM}** Normal sequential execution cannot overflow the top of the address space, because the top of memory always has the Execute Never (XN) memory attribute.
- R_{BPMP}** One or more accesses that target or wrap around the top or bottom bytes of memory, access a sequence of words at increasing memory addresses, effectively incrementing the address by four for each load or store. If this calculation overflows the top of the address space, the result is UNPREDICTABLE.
- Note, The encodings of some instructions require M, the encodings of some instructions require FP.*
- R_{ZXDN}** Where an exception entry or [tail-chaining](#) accesses bytes on the stack that span the top or bottom of the 32-bit memory address space, it is IMPLEMENTATION DEFINED whether stack limit checking is applied.

See also:

[Chapter B6 The System Address Map on page 198.](#)

B5.3 Endianness

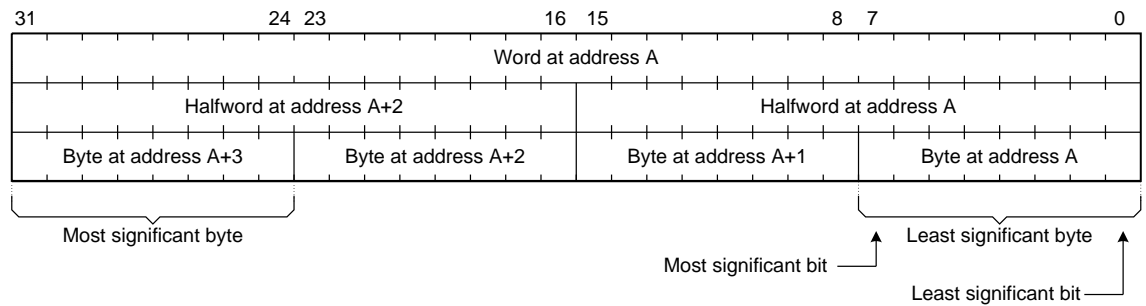
I_{CTVV}

In memory:

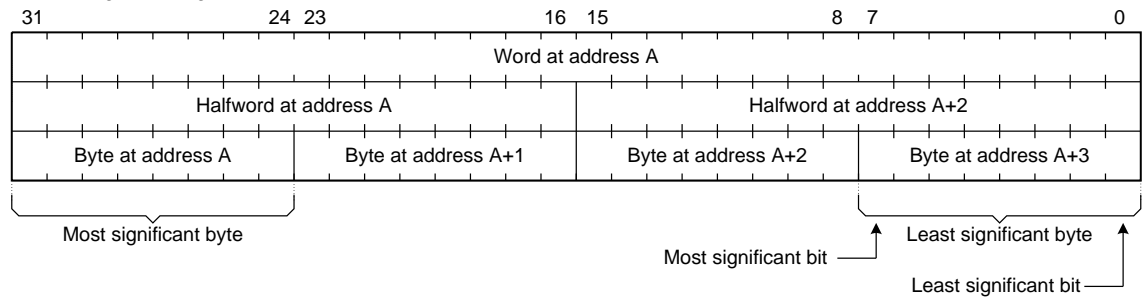
The following figures show the relationship between:

- The word at address A.
- The halfwords at addresses A and A+2.
- The bytes at addresses A, A+1, A+2, and A+3.

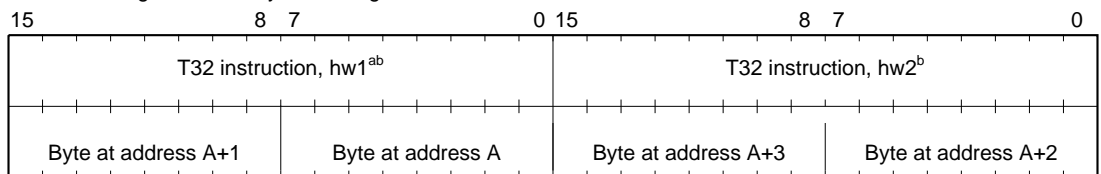
Data arranged in a little-endian format



Data arranged in a big-endian format



Instruction alignment and byte ordering



a) Bits[15:0]: this is hw 1 for a T32 instruction with a 16-bit encoding

b) Bits[31:0]: this is hw1 and hw2 for a T32 instruction with a 32-bit encoding

R_{JJQL}

Instruction fetches are always little-endian, which means that the PE assumes a little-endian arrangement of instructions in memory.

R_{MNSB} All accesses to the Private Peripheral Bus (PPB) are always little-endian, which means that the PE assumes a little-endian arrangement of the PPB registers.

R_{TFFKG} The endianness of data accesses is IMPLEMENTATION DEFINED, as indicated by [AIRCR.ENDIANNESS](#).

R_{KPCF} [AIRCR.ENDIANNESS](#) is either:

- Implemented with a static value.
- Configured by a hardware input on reset.

R_{XDJV} Instructions that cause a memory access that crosses the PPB boundary are CONstrained UNPREDICTABLE if [AIRCR.ENDIANNESS](#) is set to 1. The permitted behavior is one of the following:

- The instruction behaves as a NOP.
- The instruction raises an UNALIGNED UsageFault.
- If the instruction that crossed the PPB boundary was a load, the value of the destination register becomes UNKNOWN.
- If the instruction that crossed the PPB boundary was a store, the value of the memory locations accessed becomes UNKNOWN.

Note, a UsageFault requires M.

R_{QHWC} For data accesses, the following table shows the data element size that endianness applies to, for endianness conversion purposes.

Instruction class	Instructions	Element size
Load or store byte	LDR{S}B{T}, LDAB , LDAEXB , STLB , STLEXB , STRB{T} , TBB , LDREXB , STREXB	Byte
Load or store halfword	LDR{S}H{T}, LDAH , LDAEXH , STLH , STLEXH , and STRH{T} , TBH , LDREXH , STREXH	Halfword
Load or store word	LDR{T}, LDA , LDAEX , STL , STLEX , and STR{T} , LDREX , STREX , VLDR.F32 , VSTR.F32	Word
Load or store two words	LDRD , STRD , VLDR.F64 , VSTR.F64	Word
Load or store multiple words	LDM{IA, DB} , STM{IA, DB} PUSH (multiple registers) POP (multiple registers) , LDC , STC , VLDM VSTM , VPOP , BLX , BLXNS , BX , BXNS VLLDM , VLSTM	Word

R_{XNVS} The following instructions change the endianness of data that is loaded or stored:

- [REV](#)
Reverse word (four bytes) register, for transforming 32-bit representations.
- [REVSH](#)
Reverse halfword and sign extend, for transforming signed 16-bit representations.
- [REV16](#)
Reverse packed halfwords in a register for transforming unsigned 16-bit representations.

B5.4 Alignment behavior

- R_{LKGV}** All instruction fetches are halfword-aligned.
- R_{RQGG}** The following are unaligned data accesses that always generate an alignment fault:
- Non halfword-aligned [LDAH](#), [LDREXH](#), [LDAEXH](#), [STLH](#), [STLEXH](#), and [STREXH](#).
 - Non word-aligned [LDREX](#), [LDAEX](#), [STLEX](#), [STREX](#), [LDRD](#), [LDMIA](#), [LDMDB](#), [POP \(multiple registers\)](#), [LDC](#), [VLDR](#), [VLDM](#), [VPOP](#), [LDA](#), [STL](#), [STMIA](#), [STMDB](#), [PUSH \(multiple registers\)](#), [STC](#), [VSTR](#), [VSTM](#), [VPUSG](#), [VLLDM](#), and [VLSTM](#).
- R_{MHCM}** If [CCR.UNALIGN_TRP](#) is set to 1, the following are unaligned data accesses that generate an alignment fault:
- Non halfword-aligned [LDR{S}H{T}](#), and [STRH{T}](#).
 - Non halfword-aligned [TBH](#).
 - Non word-aligned [LDR{T}](#), and [STR{T}](#).
- R_{JLGS}** Unaligned accesses are only supported if the Main Extension is implemented.
The extension requirements are - [M](#).
- R_{WCVX}** Accesses to Device memory are always aligned.
- R_{PZTT}** If the Main Extension is not implemented, unaligned accesses generate an alignment HardFault.
The extension requirements are - [!M](#).
- R_{RNDS}** Alignment faults are synchronous and generate an UNALIGNED UsageFault.
The extension requirements are - [M](#).
- R_{BNBX}** The CONSTRAINED UNPREDICTABLE behavior of unaligned loads and stores is one of the following:
- Generate an UNALIGNED UsageFault.
 - Perform the specified load or store to the unaligned memory location.
- The extension requirements are - [M](#).*
- R_{LPVP}** Unaligned loads and stores perform the specified load and store to the unaligned memory location.

See also:

[B5.14 Normal memory](#) on page 166.

[B5.16 Device memory](#) on page 169.

B5.5 Atomicity

B5.5.1 Single-copy atomicity

- I_{NWVK}** Store operations are *single-copy atomic* if, when they overlap bytes in memory:
1. All of the writes from one of the stores are inserted into the **coherence order** of each overlapping byte.
 2. All of the writes from another of the stores are inserted into the **coherence order** of each overlapping byte.
 3. Step 2 repeats, for each single-copy store atomic operation that overlaps.
- R_{BSHJ}** The following data accesses are single-copy atomic:
- All byte accesses.
 - All halfword accesses to halfword-aligned locations.
 - All word accesses to word-aligned locations.
- R_{QNPX}** Instruction fetches are single-copy atomic at halfword granularity.
- R_{MXWC}** For instructions that access a sequence of word-aligned words, each word access is single-copy atomic.
- R_{LKPM}** For instructions that access a sequence of word-aligned words, the architecture does not require two or more subsequent word accesses to be single-copy atomic.

B5.5.2 Multi-copy atomicity

- I_{BCHK}** In a multiprocessing environment, writes to memory are *multi-copy atomic* if all of the following are true:
- All writes to the same location are observed in the same order by all observers, although some of the observers might not observe all of the writes.
 - A read of a location does not return the value of a write to that location until all observers have observed that write.
- R_{GJGP}** Writes to Normal memory are not required to be multi-copy atomic.
- R_{LBGB}** Writes to Device memory with the Gathering attribute are not required to be multi-copy atomic.
- R_{WHJR}** Writes to Device memory with the non-Gathering attribute that is single-copy atomic are also multi-copy atomic.

See also:

[B5.16 Device memory on page 169.](#)

[B5.14 Normal memory on page 166.](#)

[B5.23 Load-Acquire and Store-Release accesses to memory on page 181.](#)

B5.6 Concurrent modification and execution of instructions

- I_{TFFGC}** The Armv8 architecture limits the set of instructions that can be executed by one thread of execution as they are being modified by another thread of execution without requiring explicit synchronization.
- R_{XWVK}** Unless otherwise stated, concurrent modification and execution of instructions results in a CONSTRAINED UNPREDICTABLE choice of any behavior that can be achieved by executing any sequence of instructions from the same Security state or the same Privilege level.
- R_{BFPB}** For instructions that can be concurrently modified, the PE executes either:
- The original instruction.
 - The modified instruction.
- R_{NNQK}** A 16-bit instruction can be concurrently modified, where the 16-bit instruction before modification and the 16-bit modification is any of the following:
- [B](#).
 - [BX](#).
 - [BLX](#).
 - [BKPT](#).
 - [NOP](#).
 - [SVC](#).
- R_{KMZG}** The hw1 of a 32-bit [BL](#) immediate instruction can be concurrently modified to the most significant halfword of another [BL](#) immediate instruction.
- R_{HKGP}** The hw1 of a 32-bit [BL](#) immediate instruction can be concurrently modified to a 16-bit [B](#), [BLX](#), [BKPT](#), or [SVC](#) instruction. This modification also works in reverse.
- R_{FGBT}** The hw2 of a 32-bit [BL](#) immediate instruction can be concurrently modified to the hw2 of another [BL](#) instruction with a different immediate.
- R_{NTVD}** The hw2, of a 32-bit [B](#) immediate instruction with a condition field can be concurrently modified to the hw2 of another 32-bit [B](#) immediate instruction with a condition field with a different immediate.
- R_{CMZX}** The hw2 of a 32-bit [B](#) immediate instruction without a condition field can be concurrently modified to the hw2 of another 32-bit [B](#) immediate instruction without a condition field.

See also:

[B5.3 Endianness on page 146](#).

[B](#).

[BL](#).

BLX, BLXNS.

B5.7 Access rights

R_{KIHH} An instruction fetch or memory access is subject to the following checks in the following order:

1. Alignment.
2. SAU or IDAU, or both.
3. MPU.
4. BusFault (IBUSERR).

R_{TQJS} An exception is generated, instead of normal execution of the fetching and decoding process, if one of the following occurs.

Priority	Fault type	Cause
Highest	One of the following SecureFaults: <ul style="list-style-type: none"> • INVEP • INVTRAN 	AU violation
↓	The following MemManage fault: <ul style="list-style-type: none"> • IACCVIOL 	MPU violation
↓	The following BusFault: <ul style="list-style-type: none"> • IBUSERR 	System fault
↓	One of the following: <ul style="list-style-type: none"> • DebugMonitor exception • Halted Debug Entry 	FPB hit
↓	The following SecureFault: <ul style="list-style-type: none"> • INVEP 	SG check
↓	The following UsageFault: <ul style="list-style-type: none"> • INVSTATE 	T32 state check
Lowest	One of the following UsageFaults: <ul style="list-style-type: none"> • UNDEFINSTR • NOCP 	Undefined instruction

Note, a Secure fault requires S, a MemManage fault requires M && MPU, a Halted Debug Entry fault can only occur if Halting Debug is implemented, a DebugMonitor exception require DebugMonitor, UsageFault and BusFault require M, HardFault when !M.

R_{KPNQ} If a memory access fails its alignment check, the fetch is not presented to the SAU.

The extension requirements are - S.

R_{SDMQ} If an instruction fetch or memory access fails its AU check, the fetch is not presented to the relevant MPU for comparison.

The extension requirements are - S && MPU.

R_{FLLN} If an instruction fetch or memory access fails its MPU check, it is not issued to the memory system.

The extension requirements are - MPU.

See also:

[*B3.9 Exception numbers and exception priority numbers on page 65.*](#)

[*Chapter B8 The Armv8-M Protected Memory System Architecture on page 214.*](#)

B5.8 Observability of memory accesses

- R_{PNDH}** For a PE, the following mechanisms are treated as independent observers:
- The mechanism that performs reads from or writes to memory.
 - The mechanism that causes an instruction cache to be filled from memory or that fetches instructions to be executed directly from memory. These accesses are treated as reads.
- R_{DVFW}** The set of observers that can observe a memory access is not defined by the PE architecture, unless otherwise specified.
- I_{VSCK}** In the context of observability, *subsequent* means whichever of the following descriptions is appropriate:
- After the point in time where the location is observed by the observer.
 - After the point in time where the location is globally observed.
- R_{VCCS}** A write to a location in memory is *observed* by an observer when:
- A subsequent read of the location by the same observer would return the value that was written by the observed write or written by a write to that location by any observer that is sequenced in the coherence order of the location after the observed write.
 - A subsequent write of the location by the same observer would be sequenced in the coherence order of the location after the observed write.
- R_{XOPT}** A write to a location in memory is *globally observed* for a Shareability domain or set of observers when:
- A subsequent read of the location by any observer in that Shareability domain that is capable of observing the write would return the value that is written by the globally observed write or by a write to that location by any observer that is sequenced in the coherence order of the location after the globally observed write.
 - A subsequent write to the location by any observer in that Shareability domain would be sequenced in the coherence order of the location after the globally observed write.
- R_{RSPX}** For Device-nGnRnE memory, a read or write of a memory-mapped location in a peripheral is observed, and globally observed, only when the read or write:
- Meets the general observability conditions.
 - Can begin to affect the state of the memory-mapped peripheral.
 - Can trigger all associated side-effects, whether they affect other peripheral devices, PEs, or memory.
- R_{DGRR}** A read of a location in memory is *observed* by an observer when a subsequent write to the location by the same observer would have no effect on the value that is returned by the read.
- R_{BVJF}** A read of a location in memory is *globally observed* for a Shareability domain when a subsequent write to the location by any observer in that Shareability domain that is capable of observing the write would have no effect on the value that is returned by the read.

- R_{QRXX} Multiple writes to the same register will be observed in the same order by all observers. The architecture does not guarantee that all observers will observe all of the writes.
- R_{HMHZ} Explicit synchronization is not required on an external read or write by an external agent to be observable to a following external read or write by that agent to the same register using the same address.
- R_{TXSK} Explicit synchronization is not required for serial external accesses, either reads or writes, by a single external agent for any registers that are accessible as external system control registers.

See also:

[B5.16 Device memory on page 169.](#)

[B5.17 Device memory attributes on page 171.](#)

B5.9 Completion of memory accesses

- R_{XCTL}** A read or write is complete for a Shareability domain when the following conditions are true:
- The read or write is globally observed for that Shareability domain.
 - All instruction fetches by observers within the Shareability domain have observed the read or write.
- R_{WCMQ}** A cache or branch predictor maintenance instruction is complete for a Shareability domain when the effects of the instruction are globally observed for that Shareability domain.
- R_{SFLM}** The completion of a memory access to Device memory other than Device-nGnRnE does not guarantee the visibility of the side-effects of the access to all observers.
- R_{MWBK}** The mechanism that ensures the visibility of the side-effects of the access to all observers is IMPLEMENTATION DEFINED.

See also:

[B5.18 Shareability domains on page 174.](#)

[B5.16 Device memory on page 169.](#)

[B5.17 Device memory attributes on page 171.](#)

B5.10 Ordering requirements for memory accesses

- R_{RBDL}** Armv8-M defines access restrictions in the permitted ordering of memory accesses. These restrictions depend on the memory attributes of the accesses involved.
- R_{GJDH}** For all accesses to all memory types, the only stores by an observer that can be observed by another observer are those stores that have been [architecturally executed](#).
- R_{RXPL}** Reads and writes can be observed in any order provided that, if an [address dependency](#) exists between two reads or between a read and a write, then those memory accesses are observed in program order by all observers within the common Shareability domain of the memory addresses being accessed.
- R_{KWFG}** [Speculative writes](#) by an observer cannot be observed by another observer.
- R_{VMHG}** For Device memory with the non-Reordering attribute, memory accesses arrive at a [single peripheral](#) in program order.
- R_{WGCF}** Memory accesses caused by instruction fetches are not required to be observed in program order, unless they are separated by a [context synchronization event](#).
- R_{RJMK}** A [register data dependency](#) between the value that is returned by a load instruction and the address that is used by a subsequent memory transaction enforces an order between that load instruction and the subsequent memory transaction.

See also:

[B5.11 Ordering of implicit memory accesses on page 159.](#)

[B5.12 Ordering of explicit memory accesses on page 160.](#)

[B5.14 Normal memory on page 166.](#)

[B5.16 Device memory on page 169.](#)

[B5.18 Shareability domains on page 174.](#)

B5.11 Ordering of implicit memory accesses

R_{KPFC} There are no ordering requirements for [implicit accesses](#) to any type of memory.

See also:

[B5.1 Memory accesses on page 144.](#)

B5.12 Ordering of explicit memory accesses

- R_{EMNM} For all memory types, for accesses from a single observer, the requirements of uniprocessor semantics are maintained.
- R_{WTRP} For all types of memory, if there is a [control dependency](#) between a direct read and a subsequent direct write, the two accesses are observed in program order by any observer in the common Shareability domain of the two accesses.
- R_{XGNP} For all types of memory, if the value returned by a direct read computes data that is written by a subsequent direct write, the two accesses are observed in program order by any observer in the common Shareability domain of the two accesses.
- R_{MBNW} It is impossible for an observer to observe a write from an aligned store that both:
- Has not been executed.
 - Will not be executed.
- R_{WNRM} For an unaligned store, an observer might observe part of a store that would have completed had an exception not been taken. The store is not guaranteed to be single-copy atomic except at the byte access level.

The extension requirements are - [M](#).

See also:

[B5.1 Memory accesses](#) on page 144.

[B5.14 Normal memory](#) on page 166.

[B5.16 Device memory](#) on page 169.

[B5.17 Device memory attributes](#) on page 171.

[B5.18 Shareability domains](#) on page 174.

[B5.19 Shareability attributes](#) on page 176.

B5.13 Memory barriers

R_{WRCT} The Arm architecture supports out-of-order completion of instructions.

R_{NBQC} Armv8 supports the following memory barriers:

- *Instruction Synchronization Barrier* ([ISB](#)).
- *Data Memory Barrier* ([DMB](#)).
- *Data Synchronization Barrier* ([DSB](#)).
- *Consumption of Speculative Data Barrier* ([CSDB](#)).
- *Physical Speculative Store Bypass Barrier* ([PSSBB](#)).
- *Speculative Store Bypass Barrier* ([SSBB](#)).

R_{LOXF} The [DMB](#) and [DSB](#) memory barriers affect reads and writes to the memory system that are generated by Load/Store instructions and data or unified cache maintenance instructions that are executed by the PE. Instruction fetches are not explicit accesses.

B5.13.1 Instruction Synchronization Barrier

R_{STMG} An [ISB](#) ensures that all instructions that come after the [ISB](#) instruction in program order are fetched from the cache or memory after the [ISB](#) instruction has completed.

See also:

[InstructionSynchronizationBarrier\(\)](#).

[Context synchronization event](#)

B5.13.2 Data Memory Barrier

R_{MPSG} The required Shareability for a [DMB](#) is *Full system*, and applies to all observers in the Shareability domain.

R_{GVDL} A [DMB](#) only affects memory accesses and the operation of data cache and unified cache maintenance instructions, and has no effect on the ordering of any other instructions.

R_{HFTX} A [DMB](#) that ensures the completion of cache maintenance instructions has an access type of both loads and stores.

R_{WMRT} A [DMB](#) instruction creates two groups of memory accesses, Group A and Group B, and does not affect memory accesses that are in not in Group A or Group B:

Group A contains:

- All explicit memory accesses of the required access types from observers in the same Shareability domain as PE_e that are observed by PE_e before the [DMB](#) instruction.
- All loads of required access types from an observer PE_x in the same required Shareability domain as PE_e

that have been observed by any given different observer, PE_y, in the same required Shareability domain as PE_e before PE_y has performed a memory access that is a member of Group A.

Group B contains:

- All explicit memory accesses of the required access types by PE_e that occur in program order after the **DMB** instruction.
- All explicit memory accesses of the required access types by any given observer PE_x in the same required Shareability domain as PE_e that can only occur after a load by PE_x has returned the result of a store that is a member of Group B.

Any observer with the same required Shareability domain as PE_e observes all members of Group A before it observes any member of Group B to the extent that those group members are required to be observed, as determined by the Shareability and Cacheability of the memory addresses accessed by the group members.

If members of Group A and members of Group B access the same memory-mapped peripheral of arbitrary system-defined size, then members of Group A that are accessing Device or Normal Non-cacheable memory arrive at that peripheral before members of Group B that are accessing Device or Normal Non-cacheable memory. Where the members of Group A and Group B that are to be ordered are from the same PE, a **DMB** provides for this guarantee.

See also:

[DataMemoryBarrier\(\)](#).

[B5.18 Shareability domains on page 174.](#)

B5.13.3 Data Synchronization Barrier

I_{CNFG} The **DSB** is a memory barrier that synchronizes the execution stream with memory accesses.

R_{NKWL} The required Shareability for a **DSB** is Full system and applies to all observers in the Shareability domain.

R_{VLBF} A **DSB** instruction creates two groups of memory accesses, Group A and Group B, and does not affect memory accesses that are in not in Group A or Group B:

Group A contains:

- All explicit memory accesses of the required access types from observers in the same Shareability domain as PE_e that are observed by PE_e before the **DSB** instruction.
- All loads of required access types from an observer PE_x in the same required Shareability domain as PE_e that have been observed by any given different observer, PE_y, in the same required Shareability domain as PE_e before PE_y has performed a memory access that is a member of Group A.

Group B contains:

- All explicit memory accesses of the required access types by PE_e that occur in program order after the **DSB** instruction.
- All explicit memory accesses of the required access types by any given observer PE_x in the same required Shareability domain as PE_e that can only occur after a load by PE_x has returned the result of a store that is a member of Group B.

Any observer with the same required Shareability domain as PE_e observes all members of Group A before it observes any member of Group B to the extent that those group members are required to be observed, as determined

by the Shareability and Cacheability of the memory addresses accessed by the group members.

If members of Group A and members of Group B access the same memory-mapped peripheral of arbitrary system-defined size, then members of Group A that are accessing Device or Normal Non-cacheable memory arrive at that peripheral before members of Group B that are accessing Device or Normal Non-cacheable memory. Where the members of Group A and Group B that are to be ordered are from the same PE, a **DSB** provides for this guarantee.

R_{KMGH}

A **DSB** completes when all of the following conditions apply:

- All explicit memory accesses that are observed by PEe before the **DSB** is executed and are of the required access types, and are from observers in the same required Shareability domain as PEe, are complete for the set of observers in the required Shareability domain.
- If the required access types of the **DSB** is reads and writes, then all cache and branch predictor maintenance instructions that are issued by PEe before the **DSB** are complete for the required Shareability domain.
- All explicit accesses to the **System Control Space** that result in a context altering operation issued by PEe before the **DSB** are complete.

R_{KMBX}

No instruction that appears in program order after the **DSB** instruction can execute until the **DSB** completes.

See also:

[DataSynchronizationBarrier\(\)](#).

[B5.18 Shareability domains on page 174.](#)

B5.13.4 Consumption of Speculative Data Barrier

R_{CTSR}

The **CSDB** is a memory barrier that prevents instructions that appear in program order after the barrier completes from determining any part of the value of data derived from speculatively-executed load instructions that appeared in program order before completion of the **CSDB** memory barrier.

*The extension requirements are - **M**.*

I_{LZDK}

When a **CSDB** instruction is executed but before the **CSDB** completes and there are three instructions:

1. A load instruction speculatively executed in program order before the barrier that might or might not be architecturally executed.
2. A Conditional Move instruction that has passed its condition code check and does not have an address dependency for an input register on the speculatively-executed load.
3. A load, store, data or instruction preload appearing in program order after the barrier, which has an address dependency on the Conditional Move instruction.

The speculative execution of the load, store, data or instruction preload does not influence the allocation of cache entries to determine any part of the value of the speculatively executed load instruction by an evaluation of the cache entries which have been allocated or evicted.

*The extension requirements are - **M**.*

I_{DDTH}

When a **CSDB** instruction is executed but before the **CSDB** completes and there are three instructions:

1. A load instruction speculatively executed in program order before the barrier that might or might not be architecturally executed.

2. A Conditional Move instruction that has no dependency to pass the condition tests or for an input register on the speculatively executed load.
3. An indirect branch instruction, appearing in program order after the barrier, that is dependent on the Conditional Move instruction for the target address of the indirect branch.

The speculative execution of the indirect branch does not influence the allocation of cache entries to determine any part of the value of the speculatively executed load instruction by an evaluation of the cache entries which have been allocated or evicted.

The extension requirements are - *M*.

R_{JWCV} A CSDB instruction cannot be executed speculatively.

The extension requirements are - *M*.

I_{QZKB} A CSDB can be inserted speculatively and completed when it is known not to be speculative.

The extension requirements are - *M*.

R_{WGCX} The CSDB instruction is not available in an implementation without the Main Extension.

Note, !M.

I_{PCSF} Arm recommends that a combination of DSB SYS and an ISB is inserted to prevent consumption of speculative data.

Note, !M.

B5.13.5 Physical Speculative Store Bypass Barrier

I_{CCNK} The PSSBB prevents speculative loads from:

- Returning data older than the most recent store to the same physical address appearing in program order before the load.
- Returning data from stores using the same physical address appearing in program order after the load.

The extension requirements are - *M*.

R_{MDLZ} The PSSBB is not available in an implementation without the Main Extension.

Note, !M.

B5.13.6 Speculative Store Bypass Barrier

I_{HWND} The SSBB prevents speculative loads from:

- Returning data older than the most recent store to the same address appearing in program order before the load.
- Returning data from stores using the same address appearing in program order after the load.

The extension requirements are - *M*.

R_{MGLH} The SSBB is not available in an implementation without the Main Extension.

Note, !M.

B5.13.7 Synchronization requirements for System Control Space

- R_{SJCJ} A [DSB](#) guarantees that all writes to the System Control Space have been completed.
- R_{NPDJ} A [DSB](#) does not guarantee that the side effects of writes to the [System Control Space](#) are visible.
- R_{HMNM} A Context synchronization event guarantees that the side effects of any completed writes to the System Control Space are visible after the Context synchronization event.

See also:

[B6.3 The System Control Space \(SCS\) on page 202.](#)

B5.14 Normal memory

I_{NVRF}	Memory locations that are <i>idempotent</i> have the following properties: <ul style="list-style-type: none">• Read accesses can be repeated with no side-effects.• Repeated read accesses return the last value that is written to the resource being read.• Read accesses can fetch additional memory locations with no side-effects.• Write accesses can be repeated with no side-effects, if the contents of the location that is accessed are unchanged between the repeated writes or as the result of an exception.• Unaligned accesses can be supported.• Accesses can be merged before accessing the target memory system.
R_{QGCF}	The PE is permitted to treat regions of memory assigned the memory type Normal memory as idempotent.
R_{CGJX}	Normal memory can be marked as Cacheable or Non-cacheable. Normal memory is assigned Cacheability attributes.
R_{LCPJ}	Normal Non-cacheable memory is always treated as shareable.
R_{PKXL}	Speculative data accesses to Normal memory are permitted.
R_{WLVK}	A write to Normal memory completes in finite time.
R_{WLCV}	A write to a Non-cacheable Normal memory location reaches the endpoint for that location in the memory system in finite time.
R_{MJWF}	A completed write to Normal memory is globally observed for the <i>Shareability domain</i> in finite time without the requirement for cache maintenance instructions or memory barriers .
R_{NHFQ}	For multi-register Load/Store instructions that access Normal memory, the architecture does not define the order in which the registers are accessed.
R_{CFHV}	There is no requirement for the memory system beyond the PE to be able to identify the size of the elements accessed.

See also:

[B5.1 Memory accesses on page 144.](#)

[B5.18 Shareability domains on page 174.](#)

[B5.15 Cacheability attributes on page 168.](#)

[B5.22 Load-Exclusive and Store-Exclusive accesses to Normal memory on page 180.](#)

MAIR_ATTR, Memory Attributes Indirection Register Attributes.

B5.15 Cacheability attributes

- R_{KXJV}** The architecture provides Cacheability attributes that are defined independently for each of two conceptual levels of cache:
- The Inner cache.
 - The Outer cache.
- R_{XRWS}** The Cacheability attributes are:
- Non-cacheable.
 - Write-Through Cacheable.
 - Write-Back Cacheable.
- R_{XQXW}** It is IMPLEMENTATION DEFINED whether Write-Through Cacheable and Write-Back Cacheable can have the additional attribute Transient or Non-transient.
- I_{LDXP}** The Transient attribute is a [memory hint](#) that indicates that the benefit of caching is for a short period. The architecture does not define what is meant by a *short period*.
- R_{CFKN}** Cacheability attributes other than Non-cacheable can be complemented by the following cache allocation hints, which are independent for read and write accesses:
- Read-Allocate, Transient Read-Allocate, or No Read-Allocate.
 - Write-Allocate, Transient Write-Allocate, or No Write-Allocate.
- R_{DRTR}** The architecture does not require an implementation to make any use of cache allocation hints.
- R_{FQSS}** Any cacheable Normal memory region is treated as Read-Allocate, No Write-Allocate unless it is explicitly assigned other cache allocation hints.
- I_{FRVF}** A Cacheable location with no Read-Allocate and no Write-Allocate hints is not the same as a Non-cacheable location. A Non-cacheable location has coherency guarantees for all observers within the system that do not apply to a location that is Cacheable, no Read-Allocate, no Write-Allocate.
- R_{FTKW}** All data accesses to Non-cacheable Normal memory locations are data coherent to all observers.

See also:

[B5.14 Normal memory on page 166.](#)

B5.16 Device memory

I_{BXHS}	Device memory is a <i>memory type</i> that is assigned to regions of memory where accesses can have side-effects.
R_{WTZL}	Device memory is not cacheable.
R_{LDDN}	Device memory is always treated as shareable.
R_{PQXS}	Speculative data accesses cannot be made to Device memory. However, for instructions that access a sequence of word-aligned words, the accesses might occur multiple times.
R_{NLHC}	Speculative instruction fetches can be made to Device memory, unless the location is marked as execute-never.
R_{CSKG}	Any unaligned access to Device memory generates an UNALIGNED UsageFault exception.
R_{YMTK}	Device memory is assigned a combination of <i>Device memory attributes</i> .
R_{LFTG}	A write to Device memory completes in finite time.
R_{FSCD}	A write to a Device memory location reaches the endpoint for that location in the memory system in finite time.
R_{GTTQ}	A completed write to a Device memory location is globally observed for the Shareability domain in finite time without the requirement for cache maintenance instructions or barriers.
R_{XMCH}	If the content of a Device memory location changes without a direct write to the location, the change is observed for the Shareability domain in finite time.
R_{KJHG}	For an instruction fetch from Device memory, if a branch causes the Program Counter to point to an area of memory that is not marked as Execute-never, the implementation can either: <ul style="list-style-type: none">• Treat the fetch as if it is to a location in Normal Non-cacheable memory.• Take an IACCVIOL MemManage fault.
	<i>Note, a MemManage fault requires M.</i>
R_{DFJX}	There is no requirement for the memory system beyond the PE to be able to identify the size of the elements that are accessed, for instructions that load the following from Device memory: <ul style="list-style-type: none">• More than one general-purpose register.• One or more registers from the floating-point register file.

- R_{KVHT}** For an LDM, STM, LDRD, or STRD instruction with a register list that includes the PC, the architecture does not define the order in which the registers are accessed.
- R_{SFPK}** For an LDM, STM, VLDM, VSTM, VPOP or VPUSH instruction with a register list that does not include the PC, all registers are accessed in the order that they appear in the register list, for Device memory with the non-Reordering attribute.

See also:

[B5.1 Memory accesses on page 144.](#)

[B5.19 Shareability attributes on page 176.](#)

[B5.17 Device memory attributes on page 171.](#)

[B5.18 Shareability domains on page 174.](#)

B5.17 Device memory attributes

R_{VNSJ} Each Device memory region is assigned a combination of Device memory attributes. The attributes are:

Gathering, G and nG: The *Gathering* and *non-Gathering* attributes.

Reordering, R and nR: The *Reordering* and *non-Reordering* attributes.

Early Write Acknowledgement, E and nE: The *Early Write Acknowledgement* and *no Early Write Acknowledgement* attributes.

R_{CFFC} Each Device memory region is assigned one of the combinations in the following table:

Memory Ordering	Name	nG	nR	nE	G	R	E
Strong	Device-nGnRnE	Y	Y	Y	-	-	-
↓	Device-nGnRE	Y	Y	-	-	-	Y
↓	Device-nGRE	Y	-	-	-	Y	Y
Weak	Device-GRE	-	-	-	Y	Y	Y

R_{LJKD} Weaker memory can be accessed according to the rules specified for stronger memory:

- Memory with the:
 - G attribute can be accessed according to the rules specified for the nG attribute.
 - nG attribute cannot be accessed according to the rules specified for the G attribute.
- Memory with the:
 - R attribute can be accessed according to the rules specified for the nR attribute.
 - nR attribute cannot be accessed according to the rules specified for the R attribute.

Because the nE attribute is a hint:

- An implementation is permitted to perform an access with the E attribute in a manner consistent with the requirements specified by the nE attribute.
- An implementation is permitted to perform an access with the nE attribute in a manner consistent with the relaxations allowed by the E attribute.

R_{FJXX} For Device-GRE and Device-nGRE memory, the use of barriers is required to order accesses.

See also:

[B5.17.1 Gathering and non-Gathering Device memory attributes on page 172.](#)

[B5.17.2 Reordering and non-Reordering Device memory attributes on page 172.](#)

[B5.17.3 Early Write Acknowledgement and no Early Write Acknowledgement Device memory attributes on page 173.](#)

[B5.16 Device memory on page 169.](#)

B5.17.1 Gathering and non-Gathering Device memory attributes

G attribute

- R_{DBSX}** If multiple accesses of the same type, read or write, are to:
- The same location, with the G attribute, they can be merged into a single transaction.
 - Different locations, all with the G attribute, they can be merged into a single transaction.
- R_{KCMX}** Gathering of accesses that are separated by a memory barrier is not permitted.
- R_{JSRD}** Gathering of accesses that are generated by a Load-Acquire/Store-Release is not permitted.
- R_{MGKJ}** A read can come from intermediate buffering of a previous write if:
- The accesses are not separated by a [DMB](#) or [DSB](#) barrier.
 - The accesses are not separated by any other ordering construction that requires that the accesses are in order, for example a combination of Load-Acquire and Store-Release.
 - The accesses are not generated by a Store-Release instruction.
- I_{SRDS}** The architecture only defines programmer visible behavior. Therefore, if a programmer cannot tell whether Gathering has occurred, Gathering can be performed.

nG attribute

- R_{GVTF}** Multiple accesses to a memory location with the nG attribute cannot be merged into a single transaction.
- R_{BTWD}** A read of a memory location with the nG attribute cannot come from a cache or a buffer, but comes from the endpoint for that address in the memory system.

See also:

[B5.23 Load-Acquire and Store-Release accesses to memory on page 181.](#)

B5.17.2 Reordering and non-Reordering Device memory attributes

R attribute

- R_{RPTB}** This attribute imposes no restrictions or relaxations.

nR attribute

- R_{DFXL}** If the access is to a:
- Peripheral, it arrives at the peripheral in program order. If there is a mixture of accesses to Device nGnRE

and Device-nGnRnE in the same peripheral, these accesses occur in program order.

- Non-peripheral, this attribute imposes no restrictions or relaxations.

I_{BDWB} The IMPLEMENTATION DEFINED size of the [single peripheral](#) is the same as applies for the ordering guarantee that is provided by the [DMB](#) instruction.

R_{NDHC} The non-Reordering attribute does not require any additional ordering, other than the ordering that applies to Normal memory, between:

- Accesses with the non-Reordering attribute and accesses with the Reordering attribute.
- Accesses with the non-Reordering attribute and accesses to Normal memory.
- Accesses with the non-Reordering attribute and accesses to different peripherals of IMPLEMENTATION DEFINED size.

B5.17.3 Early Write Acknowledgement and no Early Write Acknowledgement Device memory attributes

E attribute

R_{PVSH} The E attribute imposes no restrictions or relaxations.

nE attribute

R_{FWR} Assigning the nE attribute recommends that only the endpoint of the write access returns a write acknowledgement of the access, and that no earlier point in the memory system returns a write acknowledgement.

I_{FQWQ} The E attribute is treated as a hint. Arm strongly recommends that this hint is not ignored by a PE, but is made available for use by the system.

See also:

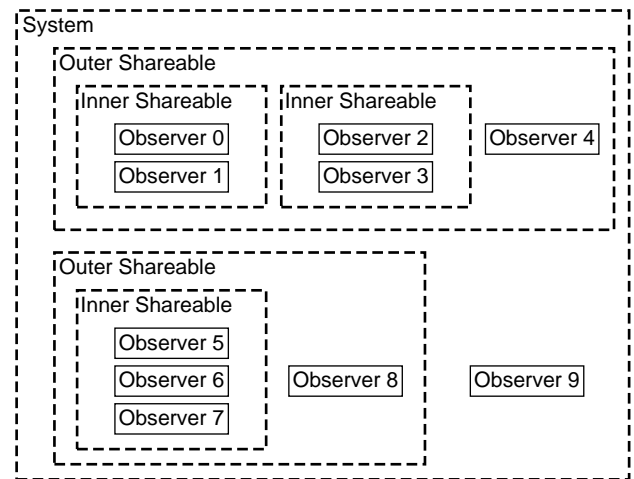
[B5.13 Memory barriers on page 161.](#)

B5.18 Shareability domains

R_{JMHL} There are two conceptual Shareability domains:

- The Inner Shareability domain.
- The Outer Shareability domain.

I_{XQWM} The following diagram shows the Shareability domains:



R_{MCPS} All observers in an Inner Shareability domain are data coherent for data accesses to memory that has the *Inner-shareable Shareability attribute*.

R_{SVCR} All observers in an Outer Shareability domain are data coherent for data accesses to memory that has the *Outer-shareable Shareability attribute*.

R_{JMFS} Each observer is a member of only a single Inner Shareability domain.

R_{BNWH} Each observer is a member of only a single Outer Shareability domain.

R_{FVBG} All members of the same Inner Shareability domain are always members of the same Outer Shareability domain.

R_{WFMV} Accesses to a shareable memory location are coherent within the Shareability domain of that location.

I_{DHJF} An Inner Shareability domain is a subset of an Outer Shareability domain, although it is not required to be a proper subset.

R_{XHJL}

Hardware is required to ensure coherency and ordering within the Shareability domain if all of the following apply:

- Before writing to a location not using the Write-Back attribute, a location in the caches that might have been written with the Write-Back attribute by an agent has been invalidated or cleaned.
- After writing the location with the Write-Back attribute, the location has been cleaned from the caches to make the write visible to external memory.
- Before reading the location with a cacheable attribute, the cache location has been invalidated, or cleaned and invalidated.
- A `DMB` barrier instruction has been executed, with a scope that applies to the common Shareability of the accesses, between any accesses to the same memory location that use different attributes.

See also:

[B5.8 Observability of memory accesses](#) on page 155.

[B5.19 Shareability attributes](#) on page 176.

B5.19 Shareability attributes

R_{CJRF} Each Normal cacheable memory region is assigned one of the following Shareability attributes:

- *Non-shareable.*
- *Inner-shareable.*
- *Outer-shareable.*

R_{PDVV} For Non-shareable memory, hardware is not required to make data accesses by different observers, other than the Debug Access Port, coherent. If a number of observers share the memory, cache maintenance instructions, in addition to the barrier operations that are required to ensure memory ordering, can ensure that the presence of caches does not lead to coherency issues.

R_{XTV D} Non-cacheable Normal memory locations are always treated as Outer Shareable.

See also:

[B5.1 Memory accesses](#) on page 144.

[B5.14 Normal memory](#) on page 166.

[B5.16 Device memory](#) on page 169.

[B5.18 Shareability domains](#) on page 174.

[B5.32 Cache maintenance operations](#) on page 192.

B5.20 Memory access restrictions

- R_{KSXT}** For accesses to any two bytes that are accessed by the same instruction, the two bytes have the same memory type and Shareability attributes, otherwise behavior is a **CONSTRAINED UNPREDICTABLE** choice of the following:
- All memory accesses that were generated by the instruction use the memory type and Shareability attributes that are associated with the first address that is accessed by the instruction.
 - All memory accesses that were generated by the instruction use the memory type and Shareability attributes that are associated with the last address that is accessed by the instruction.
 - Each memory access that is generated by the instruction uses the memory type and Shareability attribute that is associated with its own address.
 - The instruction executes as a NOP.
 - The instruction generates an alignment fault caused by the memory type.
- I_{WRBT}** Except for possible differences in cache allocation hints, Arm deprecates having different Cacheability attributes for accesses to any two bytes that are generated by the same instruction.
- R_{BFKS}** If the accesses of an instruction that cause multiple accesses to any type of Device memory cross the boundary of a memory region then the behavior is a **CONSTRAINED UNPREDICTABLE** choice of the following:
- All memory accesses that are generated by the instruction are performed as if the presence of the boundary had no effect on memory accesses.
 - All memory accesses that are generated by the instruction are performed as if the presence of the boundary had no effect on memory accesses, except that there is no guarantee of ordering between memory accesses.
 - The instruction executes as a NOP.
 - The instruction generates an alignment fault caused by the memory type.

See also:

[B5.1 Memory accesses on page 144.](#)

B5.21 Mismatched memory attributes

- R_{XHTK}** Memory locations are accessed with *mismatched attributes* if all accesses to the location do not use a common definition of all the following memory attributes of that location:
- Memory type - Device or Normal.
 - Shareability.
 - Cacheability, for the same level of the Inner or Outer cache, but excluding any cache allocation hints.
- R_{VKHJ}** When a memory location is accessed with mismatched attributes, the only permitted effects are one or more of the following:
- Uniprocessor semantics for reads and writes to that memory location might be lost. This means:
 - A read of the memory location by one agent might not return the value that was most recently written to that memory location by the same agent.
 - Multiple writes to the memory location by one agent with different memory attributes might not be ordered in program order.
 - There might be a loss of coherency when multiple agents attempt to access a memory location.
 - There might be a loss of the properties that are derived from the memory type.
 - If all Load-Exclusive/Store-Exclusive instructions that are executed across all threads to access a given memory location do not use consistent memory attributes, the exclusive monitor state becomes UNKNOWN.
 - Bytes that are written without the Write-Back cacheable attribute and that are within the same Write-Back granule as bytes that are written with the Write-Back cacheable attribute might have their values reverted to the old values as a result of cache Write-Back.
- R_{NJLB}** The loss of the properties that are associated with mismatched memory type attributes refers only to the following properties of Device memory that are additional to the properties of Normal memory:
- Prohibition of speculative read accesses.
 - Prohibition on Gathering.
 - Prohibition on Reordering.
- R_{QCKK}** If the only memory type mismatch that is associated with a memory location across all users of the memory location is between different types of Device memory, then all accesses might take the properties of the weakest Device memory type.
- R_{HCCD}** Any agent that reads a memory location with mismatched attributes using the same common definition of the Shareability and Cacheability attributes is guaranteed to access it coherently, to the extent required by that common definition of the memory attributes, only if all the following conditions are met:
- All aliases to the memory location with write permission both use a common definition of the Shareability and Cacheability attributes for the memory location, and have the Inner Cacheability attribute the same as the Outer Cacheability attribute.
 - All aliases to a memory location use a definition of the Shareability attributes that encompasses all the agents with permission to access the location.

R_{GBKH} The possible permitted effects that are caused by mismatched attributes for a memory location are defined more precisely if all the mismatched attributes define the memory location as one of:

- Any Device memory type.
- Normal Inner Non-cacheable, Outer Non-cacheable memory.

In these cases, the only permitted software-visible effects of the mismatched attributes are one or more of the following:

- Possible loss of properties that are derived from the memory type when multiple agents attempt to access the memory location.
- Possible reordering of memory transactions to the same memory location with different memory attributes, potentially leading to a loss of coherency or uniprocessor semantics. Any possible loss of coherency or uniprocessor semantics can be avoided by inserting **DMB** barrier instructions between accesses to the same memory location that might use different attributes.

R_{VVBS} If the mismatched attributes for a location mean that multiple cacheable accesses to the location might be made with different Shareability attributes, then ordering and coherency are guaranteed only if:

- Each PE that accesses the location with a cacheable attribute performs a clean and invalidate of the location before and after accessing that location.
- A **DMB** barrier with scope that covers the full Shareability of the accesses is placed between any accesses to the same memory location that use different attributes.

R_{VCXW} If multiple agents attempt to use Load-Exclusive or Store-Exclusive instructions to access a location, and the accesses from the different agents have different memory attributes associated with the location, the exclusive monitor state becomes UNKNOWN.

I_{TPWG} Arm strongly recommends that software does not use mismatched attributes for aliases of the same location. An implementation might not optimize the performance of a system that uses mismatched aliases.

See also:

[Chapter B8 The Armv8-M Protected Memory System Architecture on page 214.](#)

[B5.18 Shareability domains on page 174.](#)

[B5.15 Cacheability attributes on page 168.](#)

[B5.16 Device memory on page 169.](#)

[B5.14 Normal memory on page 166.](#)

[B5.22 Load-Exclusive and Store-Exclusive accesses to Normal memory on page 180.](#)

B5.22 Load-Exclusive and Store-Exclusive accesses to Normal memory

R_{KDWC}

For Normal memory that is:

- Non-shareable, it is IMPLEMENTATION DEFINED whether Load-Exclusive and Store-Exclusive instructions take account of the possibility of accesses by more than one observer.
- Shareable, Load-Exclusive, and Store-Exclusive instructions take account of the possibility of accesses by more than one observer.

See also:

[B5.14 Normal memory on page 166.](#)

[B5.1 Memory accesses on page 144.](#)

B5.23 Load-Acquire and Store-Release accesses to memory

I_{VVTX} The following table summarizes the Load-Acquire/Store-Release instructions.

Data type	Load- Acquire	Store- Release	Load-Acquire Exclusive	Store-Release Exclusive
32-bit word	LDA	STL	LDAEX	STLEX
16-bit halfword	LDAH	STLH	LDAEXH	STLEXH
8-bit byte	LDAB	STLB	LDAEXB	STLEXB

R_{XBRM} A Store-Release followed by a Load-Acquire is observed in program order by each observer within the Shareability domain of the memory address being accessed by the Store-Release and the memory address being accessed by the Load-Acquire.

R_{RRFK} For a Load-Acquire, observers in the Shareability domain of the address that is accessed by the Load-Acquire observe accesses in the following order:

1. The read caused by the Load-Acquire.
2. Reads and writes caused by loads and stores that appear in program order after the Load-Acquire for which the Shareability of the address that is accessed by the load or store requires that the observer observes the access.

There are no other ordering requirements on loads or stores that appear before the Load-Acquire.

R_{WLWT} For a Store-Release, observers in the Shareability domain of the address that is accessed by the Store-Release observe accesses in the following order:

1. All of the following for which the Shareability of the address that is accessed requires that the observer observes the access:
 - Reads and writes caused by loads and stores that appear in program order before the Store-Release.
 - Writes that were observed by the PE executing the Store-Release before it executed the Store-Release.
2. The write caused by the Store-Release.

There are no other ordering requirements on loads or stores that appear in program order after the Store-Release.

R_{HCKC} All Store-Release instructions are multi-copy atomic when they are observed with Load-Acquire instructions.

R_{DGXR} A Load-Acquire to an address in a memory-mapped peripheral of an arbitrary system-defined size that is defined as any type of Device memory access ensures that all memory accesses using Device memory types to the same memory-mapped peripheral that are architecturally required to be observed after the Load-Acquire will arrive at the memory-mapped peripheral after the memory access of the Load-Acquire.

R_{CKRC} A Store-Release to an address in a memory-mapped peripheral of an arbitrary system-defined size that is defined as any type of Device memory access ensures that all memory accesses using Device memory types to the same

memory-mapped peripheral that are architecturally required to be observed before the Store-Release will arrive at the memory-mapped peripheral before the memory access of the Store-Release.

- R_{GJHK}** If a Load-Acquire to a memory address in a memory-mapped peripheral of an arbitrary system-defined size that is defined as any type of Device memory access has observed the value that is stored to that address by a Store-Release, then any memory access to the memory-mapped peripheral that is architecturally required to be ordered before the memory access of the Store-Release will arrive at the memory-mapped peripheral before any memory access to the same peripheral that is architecturally required to be ordered after the memory access of the Load-Acquire.
- R_{WRLC}** Load-Acquire and Store-Release access only a single data element.
- R_{KCTN}** Load-Acquire and Store-Release accesses are single-copy atomic.
- R_{BXRP}** If a Load-Acquire or Store-Release instruction accesses an address that is not aligned to the size of the data element being accessed, the access generates an alignment fault.
- R_{NVRJ}** A Store-Release Exclusive instruction only has the release semantics if the store is successful.

See also:

[B5.18 Shareability domains on page 174.](#)

[B5.16 Device memory on page 169.](#)

B5.24 Caches

- I_{JSPB}** When a memory location is marked with a Normal Cacheable memory attribute, determining whether a copy of the memory location is held in a cache can depend on many aspects of the implementation, such as the following factors:
- The size, line length, and associativity of the cache.
 - The cache allocation algorithm.
 - Activity by other elements of the system that can access the memory.
 - Speculative instruction fetching algorithms.
 - Speculative data fetching algorithms.
 - Interrupt behaviors.
- R_{QSG}** An implementation can include multiple levels of cache, up to a maximum of seven levels, in a hierarchical memory system.
- I_{STRV}** The lower the [cache level](#), the closer the cache is to the PE.
- R_{PDSR}** Entries for addresses with a Normal cacheable attribute can be allocated to an enabled cache at any time.
- R_{JGBL}** The allocation of a memory address to a cache location is IMPLEMENTATION DEFINED.
- R_{SBGJ}** A cache entry covers at least 16 bytes and no more than 2KB of contiguous address space, aligned to its size.
- R_{XXBW}** Where a breakdown in coherency can occur, data coherency of the caches is controlled in an IMPLEMENTATION DEFINED manner.
- R_{JVJN}** The architecture cannot guarantee whether:
- A memory location that is present in the cache remains in the cache.
 - A memory location that is not present in the cache is brought into the cache.
- R_{PHWM}** If the cache is disabled, no new allocation of memory locations into the cache occurs.
- R_{LJQB}** The allocation of a memory location into a cache cannot cause the most recent value of that memory location to become invisible to an observer, if it had previously been visible to that observer.
- R_{QRLS}** If the cache is enabled, it is guaranteed that no memory location that does not have a cacheable attribute is allocated into the cache.
- R_{XXVH}** If the cache is enabled, it is guaranteed that no memory location is allocated to the cache if the access permissions

for that location are so that the location cannot be accessed by reads and cannot be accessed by writes.

- R_{SCQ}** Any cached memory location is not guaranteed to remain incoherent with the rest of memory.
- R_{ROXN}** If an implementation permits cache hits when the Cacheability control fields force all memory locations to be treated as Non-cacheable, then the cache initialization routine:
- Provides a mechanism to ensure the correct initialization of the caches.
 - Is documented clearly as part of the documentation of the device.
- In particular, if an implementation permits cache hits when the Cacheability controls force all memory locations to be treated as Non-cacheable, and the cache contents are not invalidated at reset, the initialization routine avoids any possibility of running from an uninitialized cache. It is acceptable for an initialization routine to require a fixed instruction sequence to be placed in a restricted range of memory.
- R_{WDBP}** It is UNPREDICTABLE whether the location is returned from cache or from memory when:
- The location is not marked as cacheable but is contained in the cache. This situation can occur if a location is marked as Non-cacheable after it has been allocated into the cache.
 - The location is marked as cacheable and might be contained in the cache, but the cache is disabled.
- R_{NDNN}** The architecture allows copies of control values or data values to be cached. The existence of such copies can lead to CONSTRAINED UNPREDICTABLE behavior, if the cache has not been correctly invalidated following a change of the control or data values.
- Unless explicitly stated otherwise, the behavior of the PE is consistent with:
- The old value.
 - The new value.
 - An amalgamation of the old and new values.
- I_{BMPQ}** The choice between the behaviors might, in some implementations, vary for each use of a control or data value.

See also:

[B5.25 Cache identification on page 185.](#)

[B5.28 Cache enabling and disabling on page 188.](#)

[B5.15 Cacheability attributes on page 168.](#)

[B5.29 Cache behavior at reset on page 189.](#)

[B5.33 Ordering of cache maintenance operations on page 196.](#)

[B5.21 Mismatched memory attributes on page 178.](#)

B5.25 Cache identification

- R_{WBGH}** A PE controls the implemented caches using:
- A single Cache Type Register, [CTR](#).
 - A single Cache Level ID Register, [CLIDR](#).
 - A single Cache Size Selection Register, [CSSELR](#).
 - For each implemented cache, across all levels of caching, a Cache Size Identification Register, [CCSIDR](#).
- R_{XJTL}** The number of levels of cache is IMPLEMENTATION DEFINED and can be determined from the Cache Level ID Register.
- I_{PPSB}** [Cache sets](#) and [Cache ways](#) are numbered from 0. Usually the set number is an IMPLEMENTATION DEFINED function of an address.

B5.26 Cache visibility

- R_{QLVB}** A completed write to a memory location that is Non-cacheable or Write-Through Cacheable for a level of cache made by an observer accessing the memory system inside the level of cache is visible to all observers accessing the memory system outside the level of cache without the need of explicit cache maintenance.
- R_{RCHC}** A completed write to a memory location that is Non-cacheable for a level of cache made by an observer accessing the memory system outside the level of cache is visible to all observers accessing the memory system inside the level of cache without the need of explicit cache maintenance.

See also:

[B5.15 Cacheability attributes on page 168.](#)

B5.27 Cache coherency

R_{NNDJ}

Data coherency of caches is ensured:

- When caches are not used.
- As a result of cache maintenance operations.
- By the use of hardware coherency mechanisms to ensure coherency of data accesses to memory for cacheable locations by observers in different Shareability domains.

R_{CPGW}

Hardware is not required to ensure coherency between instruction caches and memory, even for regions of memory with the Shareability attribute.

See also:

[B5.32 Cache maintenance operations on page 192.](#)

[B5.13 Memory barriers on page 161.](#)

[B5.19 Shareability attributes on page 176.](#)

B5.28 Cache enabling and disabling

- I_{PPLL}** The Configuration and Control Register, **CCR**, enables and disables caches across all levels of cache that are visible to the PE.
- R_{HRTL}** It is IMPLEMENTATION DEFINED whether the **CCR.DC** and **CCR.IC** bits affect the memory attributes that are generated by an enabled MPU.
*The extension requirements are - **M** && **MPU**.*
- I_{TNHX}** An implementation can use control bits in the Auxiliary Control Register, **ACTLR**, for finer-grained control of cache enabling.
- R_{SMDL}** For instruction fetches and data accesses, NS-Attr determines which banked instance, either Secure or Non-secure, of **CCR.IC** or **CCR.DC** is used.
- R_{DSTQ}** If the MPU is disabled, **MPU_CTRL.ENABLE** == 0, the **CCR.DC** and **CCR.IC** bits determine the cache state for cacheable regions of the default system address map.
*The extension requirements are - **M** && **MPU**.*

See also:

[B5.25 Cache identification on page 185.](#)

[B5.24 Caches on page 183.](#)

[B5.29 Cache behavior at reset on page 189.](#)

[B3.14 Secure address protection on page 81.](#)

B5.29 Cache behavior at reset

- R_{KCFK}** All caches are disabled at reset.
- R_{JMBT}** An implementation can require the use of a specific cache initialization routine to invalidate its storage array before it is enabled:
- The exact form of any required cache initialization routine is IMPLEMENTATION DEFINED.
 - If a required initialization routine is not performed, the state of an enabled cache is UNPREDICTABLE.
- R_{TVKQ}** If an implementation permits cache hits when the cache is disabled, the cache initialization routine provides a mechanism to ensure the correct initialization of the caches.
- R_{CJGV}** If an implementation permits cache hits when the cache is disabled and the cache contents are not invalidated at reset, the initialization routine avoids any possibility of running from an uninitialized cache.
- I_{JSQQ}** An initialization routine can require a fixed instruction sequence to be placed in a restricted range of memory.
- I_{JCTD}** Arm recommends that whenever an invalidation routine is required, it is based on the Armv8-M cache maintenance operations.

See also:

[B5.24 Caches on page 183.](#)

[B5.28 Cache enabling and disabling on page 188.](#)

[B5.32 Cache maintenance operations on page 192.](#)

B5.30 Behavior of Preload Data (PLD) and Preload Instruction (PLI) instructions with caches

- I_{CQLR}** PLD and PLI are memory system hints and their effect is IMPLEMENTATION DEFINED.
- I_{TPPK}** The instructions PLD and PLI do not generate exceptions but the memory system operations might generate an imprecise fault (asynchronous exception) because of the memory access.
- R_{QNGJ}** A PLD instruction does not cause any effect to the caches or memory other than the effects that, for permission or other reasons, can be caused by the equivalent load from the same location with the same context and at the same privilege level and Security state.
- R_{SFNK}** A PLD instruction does not access Device-nGnRnE or Device-nGnRE memory.
- R_{HNLN}** A PLI instruction does not cause any effect to the caches or memory other than the effects that, for permission or other reasons, can be caused by the fetch resulting from changing the PC to the location specified by the PLI instruction with the same context and at the same privilege level and Security state.
- R_{MRFG}** A PLI instruction cannot access memory that has the Device-nGnRnE or Device-nGnRE attribute.

See also:

[PLD, PLDW \(immediate\)](#).

[PLD \(literal\)](#).

[PLD, PLDW \(register\)](#).

[PLI \(immediate, literal\)](#).

[PLI \(register\)](#).

B5.31 Branch predictors

I_{GTPB} Branch predictor hardware typically uses a form of cache to hold branch information.

R_{MTBD} Branch predictors are not architecturally visible.

I_{CVCV} The [BPIALL](#) operation is provided for timing and determinism

See also:

[B5.34 Branch predictor maintenance operations on page 197.](#)

B5.32 Cache maintenance operations

I_{MRMG}	Cache maintenance operations act on particular memory locations.
R_{JJLL}	Following a Clean operation, updates made by an observer that controls the cache are made visible to other observers that can access memory at the point to which the operation is performed.
R_{VRBP}	The cleaning of a cache entry from a cache can overwrite memory that has been written by another observer only if the entry contains a location that has been written to by an observer in the Shareability domain of that memory location.
R_{SJFS}	Following an invalidate operation, updates made visible by observers that access memory at the point to which the invalidate is defined are made visible to an observer that controls the cache.
R_{PGXK}	An invalidate operation might result in the loss of updates to the locations affected by the operation that have been written by observers that access the cache.
R_{TKBD}	If the address of an entry on which the invalidate operates does not have a Normal cacheable attribute, or if the cache is disabled, then an invalidate operation ensures that this address is not present in the cache.
R_{JTXK}	If the address of an entry on which the invalidate operates has the Normal cacheable attribute, the cache invalidate operation cannot ensure that the address is not present in an enabled cache.
R_{SDVP}	A clean and invalidate operation behaves as the execution of a clean operation followed immediately by an invalidate operation. Both operations are performed to the same location.
R_{VKSN}	The clean operation cleans from the level of cache that is specified through at least the next level of cache away from the PE.
R_{GFXB}	The invalidate operation invalidates only at the level specified.
R_{KVSM}	For set/way operations and for All (entire cache) operations, the cache maintenance operation is to the next level of caching.
R_{JTWT}	For address operations, the cache maintenance operation is to the point of coherency (PoC) or to the point of unification (PoU) depending on the settings in CLIDR, {LoC, LOUU} .
R_{XLHX}	Data cache maintenance operations affect data caches and unified caches.

- R_{QKMF}** Instruction cache maintenance operations only affect instruction caches.

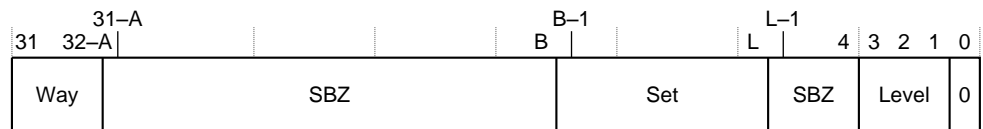
- R_{RSVL}** Cache maintenance operations are memory mapped, 32-bit write-only operations.

- R_{NSHH}** Cache maintenance operations can have one of the following side-effects:
 - Any location in the cache might be cleaned.
 - Any unlocked location in the cache might be cleaned and invalidated.

- R_{DWMM}** The ICIMVAU, DCIMVAC, DCCMVAU, DCCMVAC, and DCCIMVAC operations require the physical address in the memory map but it does not have to be cache-line aligned.

- R_{HCTC}** For DCISW, DCCSW, and DCCISW, the STR operation identifies the [cache line](#) to which it applies by specifying the following:
 - The cache set the line belongs to.
 - The way number of the line in the set.
 - The [cache level](#).

The format of the register data for a set/way operation is:



Where:

A = Log2(ASSOCIATIVITY), rounded up to the next integer if necessary.

B = (L + S).

L = Log2(LINELEN).

S = Log2(NSETS), rounded up to the next integer if necessary. ASSOCIATIVITY, LINELEN (line length, in bytes), and NSETS (number of sets) have their usual meanings and are the values for the cache level being operated on.

The values of A and S are rounded up to the next integer.

Level = ((Cache level to operate on)-1). For example, this field is 0 for operations on an L1 cache, or 1 for operations on an L2 cache.

Set = The number of the set to operate on.

Way = The number of the way to operate on.

- If L == 4 then there is no SBZ field between the set and level fields in the register.
- If A == 0 there is no way field in the register, and register bits[31:B] are SBZ.
- If the level, set, or way field in the register is larger than the size implemented in the cache, then the effect of the operation is UNPREDICTABLE.

- R_{RSBX}** After the completion of an instruction cache maintenance operation, a [context synchronization event](#) guarantees that the effects of the cache maintenance operation are visible to all instruction fetches that follow the [context synchronization event](#).
- I_{DHJQ}** Arm recommends that, wherever possible, all caches that require maintenance to ensure coherency are included in the caches affected by the architecturally-defined cache maintenance operations.
- R_{LRGS}** It is IMPLEMENTATION DEFINED whether the DCIMVAC and DCISW operations, when performed from Non-secure state either:
- Clean any data that might be Secure data before invalidating it.
 - Do not invalidate Secure data.
- The extension requirements are - S.*
- R_{VKDF}** ICIALLU, ICIMVAU, DCCMVAU, DCCMVAC, DCCSW, DCCIMVAC, DCCISW, and BPIALL operations on Secure data might be ignored if the operation was performed from Non-secure state.
- The extension requirements are - S.*
- I_{MLLC}** The following is the sequence of cache cleaning operations for a line of self-modifying code.
-
- ```

; Enter this code with <Rx> containing the new 32-bit instruction and <Ry>;
containing the address of the instruction.
; Use STRH in the first line instead of STR for a 16-bit instruction.
STR <Rx>, [<Ry>] ; Write instruction to memory
DSB ; Ensure write is visible
MOV <Rt>, 0xE000E000 ; Create pointer to base of System Control Space
STR <Ry>, [<Rt>,#0xF64] ; Clean data cache by address to point of unification
DSB ; Ensure visibility of the data cleaned from the cache
STR <Ry>, [<Rt>,#0xF58] ; Invalidate instruction cache by address to PoU
STR <Ry>, [<Rt>,#0xF78] ; Invalidate branch predictor
DSB ; Ensure completion of the invalidations
ISB ; Synchronize fetched instruction stream

```
- 
- R<sub>HXMM</sub>** If the Security attribution of memory is changed, it is IMPLEMENTATION DEFINED whether cache maintenance operations are required to keep the system state valid.
- The extension requirements are - S.*
- R<sub>JFGF</sub>** In the cache maintenance instructions that operate by Set/Way, if any index argument is larger than the value supported by the implementation, then the behavior is CONSTRAINED UNPREDICTABLE and one of the following occurs:
- The instruction generates a BusFault.
  - The instruction performs cache maintenance on one of the following:
    - No [cache lines](#).
    - A single arbitrary [cache line](#).
    - Multiple arbitrary [cache lines](#).

*Note, a BusFault requires M.*

See also:

[Cache Maintenance Operations.](#)

[Cache Maintenance Operations \(NS alias\).](#)

[B5.8 Observability of memory accesses on page 155.](#)

[B5.15 Cacheability attributes on page 168.](#)

## B5.33 Ordering of cache maintenance operations

- R<sub>GCNB</sub>** All cache and branch predictor maintenance operations that do not specify an address execute, relative to each other, in program order.
- R<sub>GXNL</sub>** All cache maintenance operations that specify an address:
- Execute in program order relative to all cache operations that do not specify an address.
  - Execute in program order relative to all cache maintenance operations that specify the same address.
  - Can execute in any order relative to cache maintenance operations that specify a different address.
- R<sub>RTJG</sub>** There is no restriction on the ordering of data or unified cache maintenance operation by address relative to any explicit load or store.
- R<sub>MJPP</sub>** There is no restriction on the ordering of a data or unified cache maintenance operation by set/way relative to any explicit load or store.
- I<sub>VXXZ</sub>** A DSB instruction can be inserted to enforce ordering as required.
- R<sub>SWBG</sub>** For the `ICIALLU` operation, the value in the register specified by the `STR` instruction that performs the operation is ignored.
- I<sub>ZQQZ</sub>** In a PE with the Security Extension, if cache maintenance operations are required when the security attribution of memory is changed, the following sequence of steps can be followed:
1. If the attribution of the address range changes from Secure to Non-secure, ensure that memory does not contain any data that is to remain secure.
  2. Execute a DSB instruction.
  3. Clean the affected lines in data or unified caches using the `DCC*` instruction.
  4. Execute a DSB instruction.
  5. Change the security attribution of the address range.
  6. Execute a DSB instruction.
  7. Invalidate the affected lines in all caches using the `DCI*` and `ICI*` instructions.
  8. Execute a Context synchronization event.

See also:

[B5.13.3 Data Synchronization Barrier on page 162.](#)

[B8.2 Security attribution on page 218.](#)

[B5.32 Cache maintenance operations on page 192.](#)



## B5.34 Branch predictor maintenance operations

- R<sub>HVXX</sub>** Branch predictor maintenance operations are independent of cache maintenance operations.
- R<sub>NSRX</sub>** A [Context synchronization event](#) event that follows a branch predictor maintenance operation guarantees that the effects of the branch predictor maintenance operation are visible to all instructions after the [Context synchronization event](#).
- R<sub>HRXF</sub>** For the `BPIALL` operation, the value in the register specified by the `STR` instruction that performs the operation is ignored.
- R<sub>LXHX</sub>** As a side-effect of a branch predictor maintenance operation, any entry in the branch predictor might be invalidated.

See also:

[Cache Maintenance Operations](#).

[Cache Maintenance Operations \(NS alias\)](#).

[BPIALL, Branch Predictor Invalidate All](#).

[B5.13 Memory barriers on page 161](#).

[DSB](#).

## Chapter B6

# The System Address Map

This chapter specifies the Armv8-M system address map rules. It contains the following sections:

[B6.1 System address map on page 199.](#)

[B6.2 The System region of the system address map on page 200.](#)

[B6.3 The System Control Space \(SCS\) on page 202.](#)

## B6.1 System address map

**R<sub>FQSD</sub>** The address space is divided into the following regions:

| Address                 | Region            | Memory type   | XN? | Cache   | Shareability  | Example usage                                                                                                         |
|-------------------------|-------------------|---------------|-----|---------|---------------|-----------------------------------------------------------------------------------------------------------------------|
| 0x00000000 - 0x1FFFFFFF | Code              | Normal        | -   | WT RA   | Non-shareable | Typically ROM or flash memory.                                                                                        |
| 0x20000000 - 0x3FFFFFFF | SRAM              | Normal        | -   | WBWA RA | Non-shareable | SRAM region typically used for on-chip RAM.                                                                           |
| 0x40000000 - 0x5FFFFFFF | Peripheral        | Device, nGnRE | XN  | -       | Shareable     | On-chip peripheral address space.                                                                                     |
| 0x60000000 - 0x7FFFFFFF | RAM               | Normal        | -   | WBWA RA | Non-shareable | Memory with write-back, write allocate cache attribute for L2 and L3 cache support.                                   |
| 0x80000000 - 0x9FFFFFFF | RAM               | Normal        | -   | WT RA   | Non-shareable | Memory with Write-Through cache attribute.                                                                            |
| 0xA0000000 - 0xBFFFFFFF | Device            | Device, nGnRE | XN  | -       | Shareable     | Peripherals accessible to all masters.                                                                                |
| 0xC0000000 - 0xDFFFFFFF | Device            | Device, nGnRE | XN  | -       | Shareable     | Peripherals accessible only to the PE.                                                                                |
| 0xE0000000 - 0xE00FFFFF | System PPB        | Device, nGnRE | XN  | -       | Shareable     | 1 MB region reserved as the PPB. This supports key resources, including the System Control Space, and debug features. |
| 0xE0100000 - 0xFFFFFFFF | System Vendor_SYS | Device, nGnRE | XN  | -       | Shareable     | Vendor System Region.                                                                                                 |

WT - Write-Through.

RA - Read-allocate.

WBWA - Write-back, write-allocate.

XN - Memory with the Execute Never memory attribute.

**I<sub>RPF</sub>** The term boundary is used to indicate the divide between memory regions stated in the system address map.

**R<sub>MRR</sub>** An access that crosses a boundary is UNPREDICTABLE. This rule also applies to the 0xFFFFFFFF - 0x00000000 boundary.

**R<sub>DNB</sub>** An unaligned or multi-word access that crosses a 0.5GB memory region boundary is UNPREDICTABLE.

See also:

[B6.2 The System region of the system address map on page 200.](#)

[B5.2 Address space on page 145.](#)

[B5.1 Memory accesses on page 144.](#)

[B5.24 Caches on page 183.](#)



Chapter B6. The System Address Map

B6.2. The System region of the system address map

**R<sub>MXRW</sub>** In a PE without the Security Extension, the Non-secure SCS is RAZ/WI and any unprivileged access to the Non-secure SCS results in a BusFault.

*The extension requirements are - **M** && **!S**. Note, if **!M** a HardFault is generated.*

**I<sub>FWLM</sub>** Arm recommends that Vendor\_SYS is divided as follows:

- 0xE0100000-0xFFFFFFFF is reserved.
- Vendor resources start at 0xF0000000.

**R<sub>DQOS</sub>** Unprivileged access to the PPB causes BusFault errors unless otherwise stated. Unprivileged accesses can be enabled to the Software Trigger Interrupt Register in the System Control Space by programming a control bit in the Configuration and Control Register.

**R<sub>RJHJ</sub>** If the exception entry context stacking, exception return context unstacking, lazy floating-point state preservation, or the stacking or unstacking of a **FNC\_RETURN** stack frame, results in an access to an address within the PPB space the behavior of the access is CONSTRAINED UNPREDICTABLE and is one of the following:

- Generates a BusFault.
- Perform the specified access to the PPB space.

This does not apply to the **VLSTM** instruction.

See also:

[B6.1 System address map on page 199.](#)

[B6.3 The System Control Space \(SCS\) on page 202.](#)

[STIR, Software Triggered Interrupt Register.](#)

[CCR, Configuration and Control Register.](#)

[B11.1.2 Debug resources on page 233.](#)

## B6.3 The System Control Space (SCS)

- R<sub>CQVK</sub>** The System Control Space (SCS) provides registers for control, configuration, and status reporting.
- R<sub>CFPK</sub>** The Secure view of the NS alias is identical to the Non-secure view of normal addresses unless otherwise stated.  
*The extension requirements are - S.*
- R<sub>GLNG</sub>** Privileged accesses to unimplemented registers are RES0.
- R<sub>NDML</sub>** Unprivileged accesses to unimplemented registers will generate a BusFault unless otherwise stated.  
*The extension requirements are - M.*
- R<sub>BMLS</sub>** The side effects of any access to the SCS that performs a context-altering operation take effect when the access completes. A *DSB* instruction can be used to guarantee completion of a previous SCS access.
- R<sub>WQOB</sub>** A *context synchronization event* guarantees that the side effects of a previous SCS access are visible to all instructions in program order following the *context synchronization event*.

See also:

[B6.2 The System region of the system address map on page 200.](#)

[System Control Block.](#)

[System Control Block \(NS alias\).](#)

[Debug Control Block.](#)

[Debug Control Block \(NS alias\).](#)

[STIR, Software Triggered Interrupt Register.](#)

[SYST\\_CSR, SysTick Control and Status Register.](#)

[Chapter B10 Nested Vectored Interrupt Controller on page 226.](#)

[Chapter B8 The Armv8-M Protected Memory System Architecture on page 214.](#)

## Chapter B7

# Synchronization and Semaphores

This chapter specifies the Armv8-M architecture rules for exclusive access instructions and non-blocking synchronization of shared memory. It contains the following sections:

[B7.1 Exclusive access instructions on page 204.](#)

[B7.2 The local monitors on page 205.](#)

[B7.3 The global monitor on page 207.](#)

[B7.4 Exclusive access instructions and the monitors on page 211.](#)

[B7.5 Load-Exclusive and Store-Exclusive instruction constraints on page 212.](#)

## B7.1 Exclusive access instructions

**R<sub>LODX</sub>** Armv8 provides non-blocking synchronization of shared memory, using synchronization primitives for accesses to both Normal and Device memory.

**R<sub>RGCP</sub>** The synchronization primitives and associated instructions are as follows:

| Function        | T32 instruction                |
|-----------------|--------------------------------|
| Load-Exclusive  |                                |
| Byte            | <a href="#">LDREXB, LDAEXB</a> |
| Halfword        | <a href="#">LDREXH, LDAEXH</a> |
| Word            | <a href="#">LDREX, LDAEX</a>   |
| Store-Exclusive |                                |
| Byte            | <a href="#">STREXB, STLEXB</a> |
| Halfword        | <a href="#">STREXH, STLEXH</a> |
| Word            | <a href="#">STREX, STLEX</a>   |
| Clear-Exclusive |                                |
|                 | <a href="#">CLREX</a>          |

**R<sub>MWFP</sub>** A Load-Exclusive instruction performs a load from memory, and:

- The executing PE marks the memory address for exclusive access.
- The local monitor of the executing PE transitions to the Exclusive Access state.

**R<sub>JHMH</sub>** The size of the marked memory block is called the *Exclusives reservation granule* (ERG), and is an IMPLEMENTATION DEFINED value that is of a power of 2 size, in the range 4 - 512 words.

**R<sub>MFTN</sub>** A marked block of the ERG is created by ignoring the least significant bits of the memory address. A marked address is any address within this marked block.

**R<sub>FMXK</sub>** In some implementations the [CTR](#) identifies the Exclusives reservation granule. Where this is not the case, the Exclusives reservation granule is treated as having the maximum of 512 words.

See also:

[B7.2 The local monitors on page 205.](#)

[B7.3 The global monitor on page 207.](#)

[B7.4 Exclusive access instructions and the monitors on page 211.](#)

[B7.5 Load-Exclusive and Store-Exclusive instruction constraints on page 212.](#)



## B7.2 The local monitors

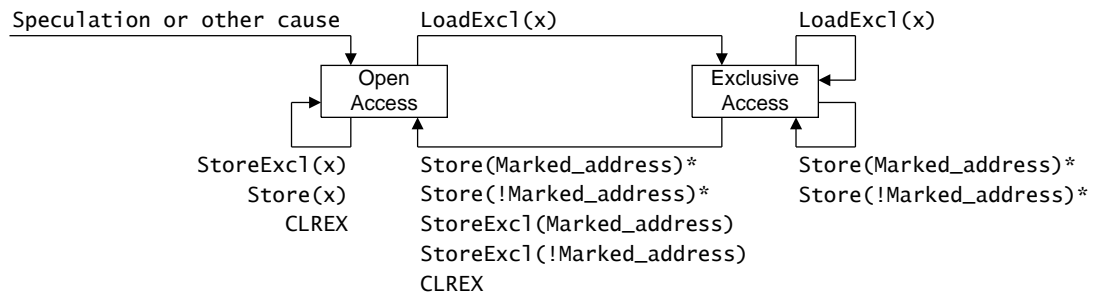
**R<sub>QTFP</sub>** Any non-aborted attempt by the same PE to use a Store-Exclusive instruction to modify any address is guaranteed to clear the marking.

**R<sub>NJWC</sub>** When a PE writes using any instruction other than a Store-Exclusive instruction:

- If the write is to a physical address that is not marked as Exclusive Access by its local monitor and that local monitor is in the Exclusive Access state, it is IMPLEMENTATION DEFINED whether the write affects the state of the local monitor.
- If the write is to a physical address that is marked as Exclusive Access by its local monitor, it is IMPLEMENTATION DEFINED whether the write affects the state of the local monitor.

**R<sub>FFFT</sub>** It is IMPLEMENTATION DEFINED whether a store to a marked physical address causes a mark in the local monitor to be cleared if that store is by an observer other than the one that caused the physical address to be marked.

**R<sub>KXNM</sub>** The state machine for the local monitor is shown here.



Operations marked \* are possible alternative IMPLEMENTATION DEFINED options.

In the diagram: `LoadExc1` represents any Load-Exclusive instruction  
`StoreExc1` represents any Store-Exclusive instruction  
`Store` represents any other store instruction.

Any `LoadExc1` operation updates the marked address to the most significant bits of the address `x` used for the operation.

The local monitor only transitions to the Exclusive Access state as the result of the architectural execution of one of the operations shown in the diagram.

Any transition of the local monitor to the Open Access state that is not caused by the architectural execution of an operation shown here does not indefinitely delay forward progress of execution.

**R<sub>WTHJ</sub>** The local monitor does not hold any physical address, but instead treats any access as matching the address of the previous Load-Exclusive instruction.

**R<sub>JWQS</sub>** A local monitor implementation can be unaware of Load-Exclusive and Store-Exclusive instructions from other PEs.

**R<sub>KJQW</sub>** The architecture does not require a load instruction by another PE that is not a Load-Exclusive instruction to have

any effect on the local monitor.

**R<sub>XMML</sub>** It is IMPLEMENTATION DEFINED whether the transition from Exclusive Access to Open Access state occurs when the `Store` or `StoreExcl` is from another observer.

**R<sub>MRSB</sub>** The architecture permits a local monitor to transition to the Open Access state as a result of speculation, or from some other cause.

**R<sub>HRHC</sub>** An exception return clears the local monitor.

See also:

[B7.4 Exclusive access instructions and the monitors on page 211.](#)

## B7.3 The global monitor

- R<sub>FKFB</sub>** For each PE in the system, the global monitor:
- Can hold at least one marked block.
  - Maintains a state machine for each marked block it can hold.
- R<sub>VDLP</sub>** For each PE, the architecture only requires global monitor support for a single marked address. Any situation that might benefit from the use of multiple marked addresses on a single PE is **CONSTRAINED UNPREDICTABLE**.
- R<sub>NNDC</sub>** The global monitor can either reside in a block that is part of the hardware on which the PE executes or exist as a secondary monitor at the memory interfaces.
- I<sub>XTLH</sub>** The **IMPLEMENTATION DEFINED** aspects of the monitors mean that the global monitor and the local monitor can be combined into a single unit, provided that the unit performs the global monitor and the local monitor functions defined in this manual.
- I<sub>KDWM</sub>** For shareable memory locations, in some implementations and for some memory types, the properties of the global monitor require functionality outside the PE. Some system implementations might not implement this functionality for all locations of memory. In particular, this can apply to:
- Any type of memory in the system implementation that does not support hardware cache coherency.
  - Non-cacheable memory, or memory treated as Non-cacheable, in an implementation that does support hardware cache coherency.
- In such a system, it is defined by the system:
- Whether the global monitor is implemented.
  - If the global monitor is implemented, which address ranges or memory types it monitors.
- I<sub>QJNL</sub>** The only memory types for which it is architecturally guaranteed that a global exclusive monitor is implemented are:
- Inner Shareable, Inner Write-Back, Outer Write-Back Normal memory with Read allocation hint and Write allocation hint and not transient.
  - Outer Shareable, Inner Write-Back, Outer Write-Back Normal memory with Read allocation hint and Write allocation hints and not transient.
- R<sub>HBKJ</sub>** The set of memory types that support atomic instructions includes all of the memory types for which a global monitor is implemented.
- R<sub>HLHS</sub>** If the global monitor is not implemented for an address range or memory type, then performing a Load-Exclusive/Store-Exclusive instruction to such a location, in the absence of any other fault, has one or more of the following effects:
- The instruction generates BusFault.
  - The instruction generates a DACCVIOL MemManage fault.
  - The instruction is treated as a NOP.

- The Load-Exclusive instruction is treated as if it were accessing a Non-shareable location, but the state of the local monitor becomes UNKNOWN.
- The Store-Exclusive instruction is treated as if it were accessing a Non-shareable location, but the state of the local monitor becomes UNKNOWN.
- The value held in the result register of the Store-Exclusive instruction becomes UNKNOWN.

*Note, a MemManage Fault requires M & MPU, a BusFault requires M.*

**R<sub>FQRT</sub>** For write transactions generated by non-PE observers that do not implement exclusive accesses or other atomic access mechanisms, the effect that writes have on the global monitor and the local monitor that are used by an Arm PE is IMPLEMENTATION DEFINED. The writes might not clear the global monitors of other PEs for:

- Some address ranges.
- Some memory types.

### B7.3.1 Load-Exclusive and Store-Exclusive

**R<sub>RXVB</sub>** The global monitor only supports a single outstanding exclusive access to shareable memory for each PE.

**R<sub>GXLF</sub>** The architecture does not require a load instruction by another PE, that is not a Load-Exclusive instruction, to have any effect on the global monitor.

**R<sub>MPKM</sub>** A Load-Exclusive instruction by one PE has no effect on the global monitor state for any other PE.

**R<sub>MFGC</sub>** A Store-Exclusive instruction performs a conditional store to memory:

- The store is guaranteed to succeed only if the physical address accessed is marked as exclusive access for the requesting PE and both the local monitor and the global monitor state machines for the requesting PE are in the Exclusive Access state. In this case:
  - A status value of 0 is returned to a register to acknowledge the successful store.
  - The final state of the global monitor state machine for the requesting PE is IMPLEMENTATION DEFINED.
  - If the address accessed is marked for exclusive access in the global monitor state machine for any other PE then that state machine transitions to Open Access state.
- If no address is marked as exclusive access for the requesting PE, the store does not succeed:
  - A status value of 1 is returned to a register to indicate that the store failed.
  - The global monitor is not affected and remains in Open Access state for the requesting PE.
- If a different physical address is marked as exclusive access for the requesting PE, it is IMPLEMENTATION DEFINED whether the store succeeds or not:
  - If the store succeeds a status value of 0 is returned to a register, otherwise a value of 1 is returned.
  - If the global monitor state machine for the PE was in the Exclusive Access state before the Store-Exclusive instruction it is IMPLEMENTATION DEFINED whether that state machine transitions to the Open Access state.

**R<sub>NNMG</sub>** In a shared memory system, the global monitor implements a separate state machine for each PE in the system. The state machine for accesses to shareable memory by PE(n) can respond to all the shareable memory accesses visible to it.

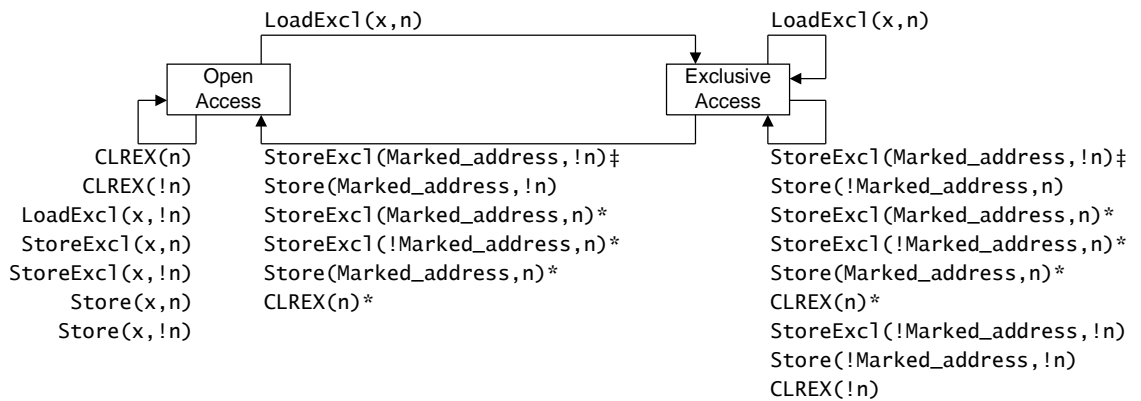
**R<sub>WKPJ</sub>** In a shared memory system, the global monitor implements a separate state machine for each observer that can generate a Load-Exclusive or a Store-Exclusive instruction in the system.

**R<sub>NWWH</sub>** Whenever the global monitor state for a PE changes from Exclusive access to Open access, an event is generated and held in the Event register for that PE. This register is used by the Wait for Event mechanism.

### B7.3.2 Load-Exclusive and Store-Exclusive in Shareable memory

**R<sub>HKQT</sub>** A Load-Exclusive instruction from shareable memory performs a load from memory, and causes the physical address of the access to be marked as exclusive access for the requesting PE. This access can also cause the exclusive access mark to be removed from any other physical address that has been marked by the requesting PE.

**R<sub>GDMJ</sub>** The state machine for PE(n) in a global monitor is as follows.



‡StoreExc1(Marked\_address, !n) clears the monitor only if the StoreExc1 updates memory

Operations marked \* are possible alternative IMPLEMENTATION DEFINED options.

In the diagram: LoadExc1 represents any Load-Exclusive instruction

StoreExc1 represents any Store-Exclusive instruction

Store represents any other store instruction.

Any LoadExc1 operation updates the marked address to the most significant bits of the address x used for the operation.

**R<sub>RGFK</sub>** Whether a Store-Exclusive instruction successfully updates memory or not depends on whether the address accessed matches the marked shareable memory address for the PE issuing the Store-Exclusive instruction, and whether the local monitor and the global monitor are in the exclusive state.

**R<sub>QVWF</sub>** When the global monitor is in the Exclusive Access state, it is IMPLEMENTATION DEFINED whether a CLREX instruction causes the global monitor to transition from Exclusive Access to Open Access state.

**R<sub>DLMP</sub>** A Load-Exclusive instruction can only update the marked shareable memory address for the PE issuing the Load-Exclusive instruction.

**R<sub>B<sub>SG</sub>B</sub>**

It is IMPLEMENTATION DEFINED:

- Whether a modification to a Non-shareable memory location can cause a global monitor to transition from Exclusive Access to Open Access state.
- Whether a Load-Exclusive instruction to a Non-shareable memory location can cause a global monitor to transition from Open Access to Exclusive Access state.

See also:

[B7.4 Exclusive access instructions and the monitors on page 211.](#)

## B7.4 Exclusive access instructions and the monitors

**R<sub>VXWN</sub>** The Store-Exclusive instruction defines the register to which the status value of the monitors is returned.

**R<sub>DTRN</sub>** A Store-Exclusive instruction performs a conditional store to memory that depends on the state of the local monitor:

- **If the local monitor is in the Exclusive Access state:**

- If the address of the Store-Exclusive instruction is the same as the address that has been marked in the monitor by an earlier Load-Exclusive instruction, then the store occurs. Otherwise, it is IMPLEMENTATION DEFINED whether the store occurs.
- A status value is returned to a register:
  - \* If the store took place the status value is 0.
  - \* Otherwise, the status value is 1.
- The local monitor of the executing PE transitions to the Open Access state.

- **If the local monitor is in the Open Access state:**

- No store takes place.
- A status value of 1 is returned to a register.
- The local monitor remains in the Open Access state.

**R<sub>DFNB</sub>** A Store-Exclusive instruction performs a store to Shareable memory that depends on the state of both the local monitor and the global monitor:

- **If both the local monitor and the global monitor are in the Exclusive Access state:**

- If the address of the Store-Exclusive instruction is the same as the address that has been marked in the monitor by an earlier Load-Exclusive instruction, then the store occurs. Otherwise, it is IMPLEMENTATION DEFINED whether the store occurs.
- A status value is returned to a register:
  - \* If the store took place the status value is 0.
  - \* Otherwise, the status value is 1.
- The local monitor of the executing PE transitions to the Open Access state.

- **If either the local monitor or the global monitor is in the Open Access state:**

- No store takes place.
- A status value of 1 is returned to a register.
- The local monitor of the executing PE transitions to the Open Access state.
- The global monitor that is associated with the executing PE transitions to the Open Access state.

See also:

[B7.2 The local monitors on page 205.](#)

[B7.3 The global monitor on page 207.](#)

## B7.5 Load-Exclusive and Store-Exclusive instruction constraints

|                         |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                  |
|-------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>I<sub>RTHW</sub></b> | The Load-Exclusive and Store-Exclusive instructions are intended to work together as a pair, for example a LDREX/STREX pair or a LDREXB/STREXB pair.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |
| <b>R<sub>BHPN</sub></b> | The architecture does not require an address or size check as part of the <code>IsExclusiveLocal()</code> function.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                              |
| <b>R<sub>LHLG</sub></b> | If two <code>StoreExcl</code> instructions are executed without an intervening <code>LoadExcl</code> instruction the second <code>StoreExcl</code> instruction returns a status value of 1.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                      |
| <b>R<sub>DVRQ</sub></b> | The architecture does not require every <code>LoadExcl</code> instruction to have a subsequent <code>StoreExcl</code> instruction.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                               |
| <b>R<sub>JXXS</sub></b> | If the transaction size of a <code>StoreExcl</code> instruction is different from the preceding <code>LoadExcl</code> instruction in the same thread of execution, behavior is a CONSTRAINED UNPREDICTABLE choice of: <ul style="list-style-type: none"><li>• The <code>StoreExcl</code> either passes or fails, and the status value returned by the <code>StoreExcl</code> is UNKNOWN.</li><li>• The block of data of the size of the larger of the transaction sizes used by the <code>LoadExcl/StoreExcl</code> pair at the address accessed by the <code>LoadExcl/StoreExcl</code> pair, is UNKNOWN.</li></ul>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                              |
| <b>R<sub>GVWN</sub></b> | The hardware only ensures that a <code>LoadExcl/StoreExcl</code> pair succeeds if the <code>LoadExcl</code> and the <code>StoreExcl</code> have the same transaction size.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                       |
| <b>R<sub>XLSK</sub></b> | Forward progress can only be made using <code>LoadExcl/StoreExcl</code> loops if, for any <code>LoadExcl/StoreExcl</code> loop within a single thread of execution if both of the following are true: <ul style="list-style-type: none"><li>• There are no explicit memory accesses, pre-loads, direct or indirect register writes, cache maintenance instructions, <code>SVC</code> instructions, or exception returns between the Load-Exclusive and the Store-Exclusive.</li><li>• The following conditions apply between the Store-Exclusive having returned a fail result and the retry of the Load-Exclusive:<ul style="list-style-type: none"><li>– There are no stores to any location within the same Exclusives reservation granule that the Store-Exclusive is accessing.</li><li>– There are no direct or indirect register writes, other than changes to the flag fields in <code>APSR</code> or <code>FPSCR</code>, caused by data processing or comparison instructions.</li><li>– There are no direct or indirect cache maintenance instructions, <code>SVC</code> instructions, or exception returns.</li></ul></li></ul> <p>The exclusive monitor can be cleared at any time without an application-related cause, provided that such clearing is not systematically repeated so as to prevent the forward progress in finite time of at least one of the threads that is accessing the exclusive monitor.</p> |
| <b>I<sub>RFXR</sub></b> | Keeping the <code>LoadExcl</code> and the <code>StoreExcl</code> operations close together in a single thread of execution minimizes the chance of the exclusive monitor state being cleared between the <code>LoadExcl</code> instruction and the <code>StoreExcl</code> instruction. Therefore, for best performance, Arm strongly recommends a limit of 128 bytes between <code>LoadExcl</code> and <code>StoreExcl</code> instructions in a single thread of execution.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                      |



- R<sub>PKQF</sub>** The architecture sets an upper limit of 2048 bytes on the Exclusives reservation granule that can be marked as exclusive.
- I<sub>PGGN</sub>** For performance reasons, Arm recommends that objects that are accessed by exclusive accesses are separated by the size of the exclusive reservations granule.
- R<sub>XPDN</sub>** After taking a BusFault or a MemManage fault, the state of the exclusive monitors is UNKNOWN.  
*The extension requirements are - M.*
- R<sub>FCRN</sub>** For the memory location accessed by a LoadExcl/StoreExcl pair, if the memory attributes for a StoreExcl instruction are different from the memory attributes for the preceding LoadExcl instruction in the same thread of execution, behavior is CONSTRAINED UNPREDICTABLE.
- R<sub>DMJW</sub>** The effect of a data or unified cache invalidate, clean, or clean and invalidate instruction on a local exclusive monitor or a global exclusive monitor that is in the Exclusive Access state is CONSTRAINED UNPREDICTABLE, and the instruction might clear the monitor, or it might leave it in the Exclusive Access state. For address-based maintenance instructions, this also applies to the monitors of other PEs in the same Shareability domain as the PE executing the cache maintenance instruction, as determined by the Shareability domain of the address being maintained.
- I<sub>MDHL</sub>** Arm strongly recommends that implementations ensure that the use of such maintenance instructions by a PE in the Non-secure state cannot cause a denial of service on a PE in the Secure state.
- R<sub>RRTJ</sub>** In the event of repeatedly-contending LoadExcl/StoreExcl instruction sequences from multiple PEs, an implementation ensures that forward progress is made by at least one PE.

## Chapter B8

# The Armv8-M Protected Memory System Architecture

This chapter specifies the Armv8-M *Protected Memory System Architecture* (PMSAv-8) rules, and in particular the rules for the optional *Memory Protection Unit*(MPU) and the optional *Security Attribution Unit* (SAU). It contains the following sections:

[B8.1 \*Memory Protection Unit\* on page 215.](#)

[B8.2 \*Security attribution\* on page 218.](#)

[B8.3 \*Security attribution unit \(SAU\)\* on page 221.](#)

[B8.4 \*IMPLEMENTATION DEFINED Attribution Unit \(IDAU\)\* on page 222.](#)

## B8.1 Memory Protection Unit

- R<sub>HPNK</sub>** In an implementation that includes the Protected Memory System Architecture (PMSA), system address space is protected by a Memory Protection Unit (MPU).  
*The extension requirements are - MPU.*
- R<sub>TBPJ</sub>** PMSAv8-M only supports a unified memory model. All enabled regions support instruction and data accesses.  
*The extension requirements are - MPU.*
- R<sub>HBNG</sub>** Memory attributes are determined from the default system address map or by using an MPU.  
*The extension requirements are - MPU.*
- R<sub>BXCN</sub>** MPU support in Armv8-M is optional.  
*The extension requirements are - MPU.*
- R<sub>MCCL</sub>** The default memory map can be configured to provide a background region for privileged accesses.  
*The extension requirements are - MPU.*
- R<sub>JVJC</sub>** When the MPU is disabled or not present, accesses use memory attributes from the default system address map.  
*The extension requirements are - !MPU.*
- R<sub>KLHL</sub>** If the MPU is enabled, attributes for memory accesses that hit in a single region are provided by the hit region.  
*The extension requirements are - MPU.*
- R<sub>DDBM</sub>** The MPU divides the memory into regions.  
*The extension requirements are - MPU.*
- R<sub>JVCN</sub>** An individual MPU region is defined by:  
`Address >= MPU_RBAR.BASE:'00000' && Address <= MPU_RLAR.LIMIT:'11111'`  
*The extension requirements are - MPU.*
- R<sub>MNDS</sub>** The number of supported MPU regions is IMPLEMENTATION DEFINED.  
*The extension requirements are - MPU.*
- I<sub>WTCL</sub>** Because the **MPU\_TYPE** register is banked, an implementation can have a different number of MPU regions, including no MPU regions, for each Security state.  
*The extension requirements are - S && MPU.*
- R<sub>XGFK</sub>** All MPU regions are aligned to a multiple of 32 bytes.  
*The extension requirements are - MPU.*
- R<sub>BPGB</sub>** The PE can fetch and execute instructions from each MPU region according to the value of **MPU\_RBAR.XN**.  
*The extension requirements are - MPU.*
- R<sub>NBPN</sub>** Accesses to the following region of memory `0xE0000000-0xE00FFFFFF`, the *Private Peripheral Bus* (PPB) always use memory attributes from the default system address map.  
*The extension requirements are - MPU.*

|                         |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                   |
|-------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>R<sub>ZLHD</sub></b> | Unless otherwise stated, all load, store, and instruction fetch transactions are subject to an MPU check.<br><i>The extension requirements are - MPU.</i>                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |
| <b>R<sub>DNXT</sub></b> | If <b>MPU_CTRL.ENABLE</b> is zero, MPU checks are carried out against the default system address map and not against any defined MPU regions.<br><i>The extension requirements are - MPU.</i>                                                                                                                                                                                                                                                                                                                                                                                                                                     |
| <b>I<sub>HSCD</sub></b> | The MPU check is one of a number of checks carried out on any load, store or instruction fetch transaction including alignment, security attribution checks, and a check for any BusFaults.<br><i>The extension requirements are - MPU.</i>                                                                                                                                                                                                                                                                                                                                                                                       |
| <b>R<sub>VHHL</sub></b> | Exception vector reads from the Vector Address Table always use the default system address map and are not subject to an MPU check.<br><i>The extension requirements are - MPU.</i>                                                                                                                                                                                                                                                                                                                                                                                                                                               |
| <b>R<sub>VWXJ</sub></b> | If <b>MPU_CTRL.HFNMIENA</b> is set to 0, any load, store or instruction fetch transaction where the requested execution priority is negative will use the Default Address Map for MPU checks.<br><i>The extension requirements are - MPU.</i>                                                                                                                                                                                                                                                                                                                                                                                     |
| <b>R<sub>VPJQ</sub></b> | For the purposes of MPU checks, any change in the current execution priority resulting from a <b>MSR</b> or <b>CPS</b> requires a <b>Context synchronization event</b> to guarantee visibility.<br><i>The extension requirements are - MPU.</i>                                                                                                                                                                                                                                                                                                                                                                                   |
| <b>R<sub>TGQD</sub></b> | Any load, store or instruction fetch transactions to the PPB, within the range <b>0xE0000000-0xE0FFFFFF</b> , are not subject to an MPU check but are checked against the default system address map. Instruction fetches to this region generate an XN MemManage fault.<br><i>The extension requirements are - MPU.</i>                                                                                                                                                                                                                                                                                                          |
| <b>R<sub>LLLP</sub></b> | Any MPU region lookup performed for a load, store or instruction fetch transaction will generate a precise MemManage Fault if any of the following is true: <ul style="list-style-type: none"><li>• The address accessed by the load, store or instruction fetch transaction matches more than one MPU region.</li><li>• The load, store or instruction fetch transaction does not match all of the access conditions for the MPU region being accessed.</li><li>• The load, store or instruction fetch transaction matches a background region or the default memory map.</li></ul> <i>The extension requirements are - MPU.</i> |
| <b>R<sub>KDJG</sub></b> | The MPU is restricted in how it can change the default memory map attributes associated with System space, that is, for addresses in the region <b>0xE0100000-0xFFFFFFFF</b> . Unless otherwise stated, system space is always XN (Execute Never) and it is always Device-nGnR. If the MPU maps this to a type other than Device-nGnRnE, it is UNKNOWN whether the region is treated as Device-nGnRE or as Device-nGnRnE.<br><i>The extension requirements are - MPU.</i>                                                                                                                                                         |
| <b>R<sub>KMTF</sub></b> | Unless otherwise stated for data accesses, the MPU memory attribution and privilege checking uses the configuration registers that correspond to the current Security state of the PE.<br><i>The extension requirements are - MPU &amp;&amp; S.</i>                                                                                                                                                                                                                                                                                                                                                                               |
| <b>R<sub>RLBR</sub></b> | For instruction fetches, the MPU memory attribution and privilege checking uses the configuration registers associated with the security of the target address.<br><i>The extension requirements are - MPU.</i>                                                                                                                                                                                                                                                                                                                                                                                                                   |

**R<sub>PLJG</sub>** Setting [MPU\\_CTRL.HFNMIENA](#). to zero disables the MPU if the requested priority for the handler of the HardFault, NMI and exceptions that the MPU is associated with is negative.

*The extension requirements are - [MPU](#).*

**R<sub>MKJC</sub>** Setting the [MPU\\_RNR.REGION](#) field to a value that does not correspond to an implemented memory region is CONSTRAINED UNPREDICTABLE as follows:

- Any subsequent read of [MPU\\_RNR.REGION](#) returns an UNKNOWN value.
- Any read of a register that is in an unimplemented region returns an UNKNOWN value.
- Any write to a register indirected by [MPU\\_RNR.REGION](#) causes all state that is indirected by that register to become UNKNOWN.

*The extension requirements are - [MPU](#).*

See also:

[B6.1 System address map](#) on page 199.

[B5.7 Access rights](#) on page 153.

[B5.17 Device memory attributes](#) on page 171.

[B5.19 Shareability attributes](#) on page 176.

[B5.20 Memory access restrictions](#) on page 177.

[B5.21 Mismatched memory attributes](#) on page 178.

[B5.22 Load-Exclusive and Store-Exclusive accesses to Normal memory](#) on page 180.

[B5.23 Load-Acquire and Store-Release accesses to memory](#) on page 181.

[MPU\\_CTRL](#), [MPU Control Register](#).

[TT\\_RESP](#), [Test Target Response Payload](#).

## B8.2 Security attribution

**I<sub>SBSJ</sub>** The Secure Attribution Unit and the Implementation Defined Attribution Unit are collectively referred to as the Attribution Unit (AU).

*The extension requirements are - S.*

**R<sub>JGHS</sub>** The Security Extension defines three levels of memory security attribution. In ascending order of security, these are:

1. Non-secure.
2. Secure and Non-secure callable.
3. Secure and not Non-secure callable.

*The extension requirements are - S.*

**R<sub>RPKG</sub>** The following units can provide security attribution information:

- A *Security attribution unit* (SAU) inside the PE.
- An IMPLEMENTATION DEFINED *attribution unit* (IDAU) external to the PE. The presence of such a unit is IMPLEMENTATION DEFINED.

*The extension requirements are - S.*

**R<sub>MGXN</sub>** The attribution information from the SAU is used unless the IDAU specifies attributes with a higher security, in which case the IDAU attributes override the SAU attributes. This rule does not apply to architecturally defined ranges exempt from memory attribution.

*The extension requirements are - S.*

**R<sub>NJGR</sub>** An *attribution unit* (AU) violation is defined as being a violation raised by either the SAU or the IDAU.

*The extension requirements are - S.*

**R<sub>QGVS</sub>** All boundaries between address ranges with different security attributes are aligned to 32-byte boundaries.

*The extension requirements are - S.*

**R<sub>BLJT</sub>** The behavior of the following address ranges is fixed, so they are exempt from memory attribution by both the SAU and IDAU:

**0xF0000000 – 0xFFFFFFFF**

If the PE implements the Security Extension, this memory range is always marked as Secure and not Non-secure callable for instruction fetches.

If the Security Extension is not present, this range is marked as Non-secure.

### Ranges exempt from checking security violation

The following address ranges are marked with the Security state indicated by NS-Req, that is, the current state of the PE for non-debug accesses. This marking sets the NS-Attr to NS-Req:

0xE0000000 - 0xE0002FFF: ITM, DWT, FPB.

0xE000E000 - 0xE000EFFF: SCS Secure and Non-secure range.

0xE002E000 - 0xE002EFFF: SCS Non-Secure alias range.

0xE0040000 - 0xE0041FFF: TPIU, ETM.

0xE00FF000 - 0xE00FFFFF: ROM table.

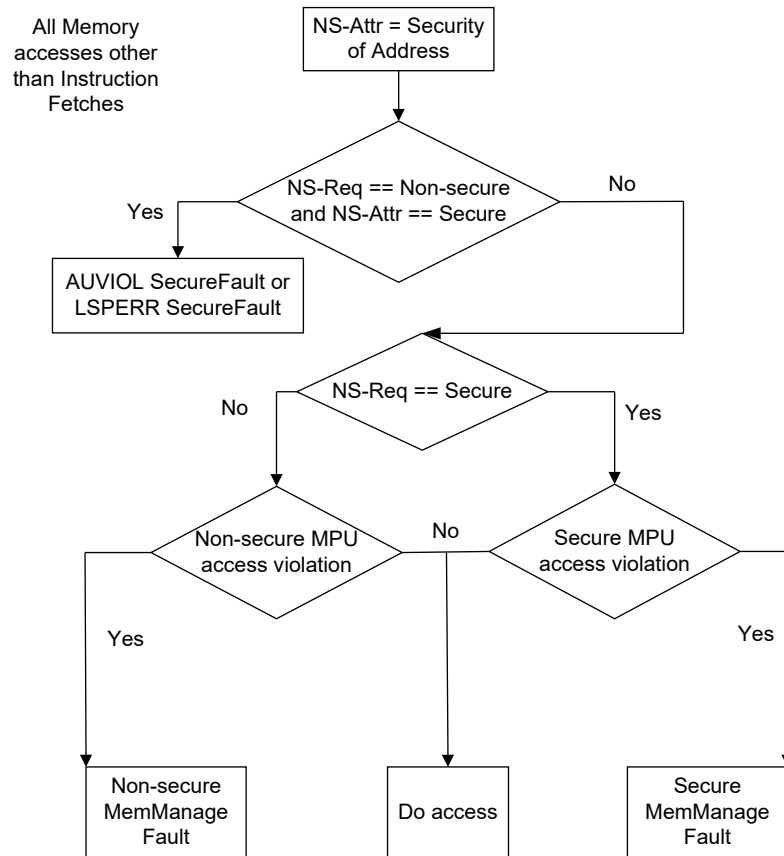
0xE0000000 - 0xFFFFFFFF: for instruction fetch only.

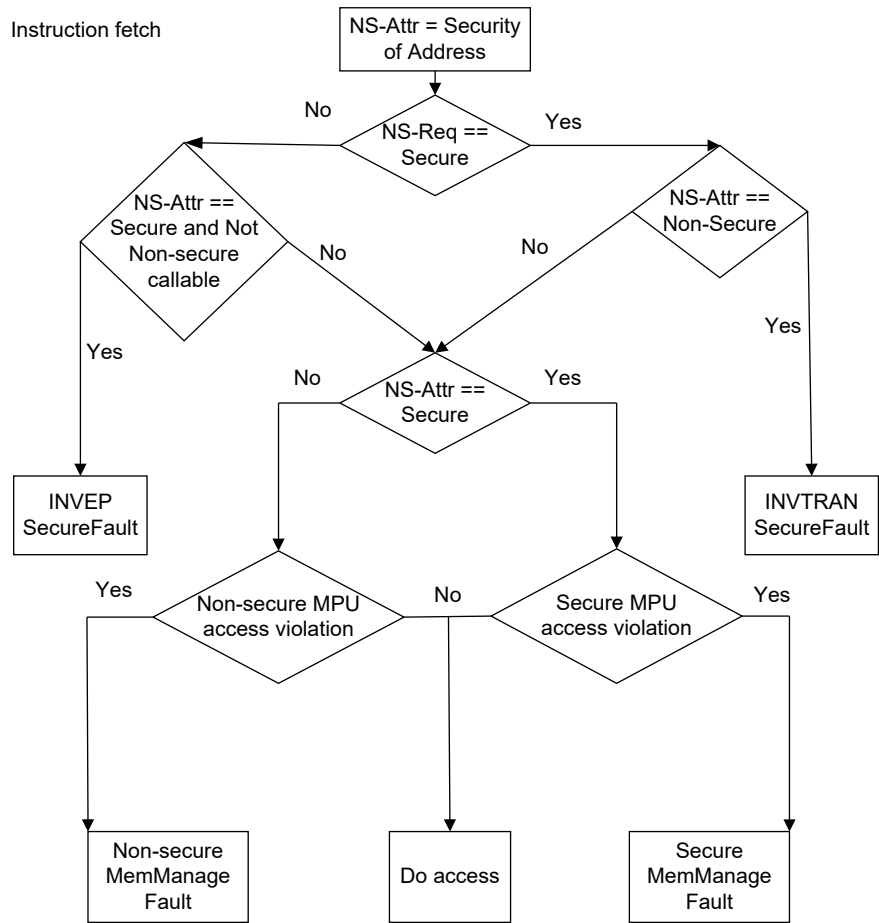
Additional address ranges specified by the IDAU.

The extension requirements are - **S**. Note, some address ranges require DB.

I<sub>VPWL</sub>

The Security attribution and MPU check sequence, for all data accesses which are not instruction fetches and accesses for instruction fetches are shown in the following diagrams.





If the memory transaction is lazy state preservation then NS-req is determined by [FPCCR.S](#). If the memory transaction is an exception vector fetch access NS-Req is set to the Security state of the exception. Unless otherwise specified, in all other cases it is the current Security state.

The extension requirements are - [S](#).

See also:

[B8.3 Security attribution unit \(SAU\) on page 221.](#)

[B8.4 IMPLEMENTATION DEFINED Attribution Unit \(IDAU\) on page 222.](#)



## B8.3 Security attribution unit (SAU)

**R<sub>VFLR</sub>** The SAU configuration defines an IMPLEMENTATION DEFINED number of memory regions. The number of regions is indicated by [SAU\\_TYPE.SREGION](#).

*The extension requirements are - S.*

**I<sub>PPLK</sub>** The memory regions defined by the SAU configuration are referred to as SAU\_REGION<sub>n</sub>, where n is a number from 0 - ([SAU\\_TYPE.SREGION](#)-1).

*The extension requirements are - S.*

**R<sub>RVFP</sub>** The SAU region configuration fields can only be accessed indirectly using the window registers, [SAU\\_RNR](#) shown in the following table.

| SAU region configuration field  | Associated window register field |
|---------------------------------|----------------------------------|
| SAU_REGION <sub>n</sub> .ENABLE | <a href="#">SAU_RLAR.ENABLE</a>  |
| SAU_REGION <sub>n</sub> .NSC    | <a href="#">SAU_RLAR.NSC</a>     |
| SAU_REGION <sub>n</sub> .BADDR  | <a href="#">SAU_RBAR.BADDR</a>   |
| SAU_REGION <sub>n</sub> .LADDR  | <a href="#">SAU_RLAR.LADDR</a>   |

*The extension requirements are - S.*

**R<sub>KRSC</sub>** When the SAU is enabled, an address is defined as matching a region in the SAU if the following is true:

Address >= SAU\_REGION<sub>n</sub>.BADDR: '00000' && Address <= SAU\_REGION<sub>n</sub>.LADDR: '11111'.

*The extension requirements are - S.*

**R<sub>MPJC</sub>** Memory is marked as Secure by default. However, if the address matches a region with SAU\_REGION<sub>n</sub>.ENABLE set to 1 and SAU\_REGION<sub>n</sub>.NSC set to 0, then memory is marked as Non-secure.

*The extension requirements are - S.*

**R<sub>WGDK</sub>** An address that matches multiple SAU regions is marked as Secure and not Not-secure callable regardless of the attributes specified by the regions that matched the address.

*The extension requirements are - S.*

**R<sub>GVFQ</sub>** When the SAU is not enabled:

- Addresses are not checked against the SAU regions.
- The attribution of the address space is determined by the [SAU\\_CTRL.ALLNS](#) field.

*The extension requirements are - S.*

**R<sub>MBJN</sub>** To permit lockdown of the SAU configuration, it is IMPLEMENTATION DEFINED whether [SAU\\_RLAR](#), [SAU\\_RBAR](#), [SAU\\_CTRL](#), and [SAU\\_RNR](#) are writable.

*The extension requirements are - S.*

**R<sub>BBCT</sub>** Setting the [SAU\\_RNR.REGION](#) field to a value that does not correspond to an implemented memory region is CONSTRAINED UNPREDICTABLE as follows:

- Any subsequent read of [SAU\\_RNR.REGION](#) returns an UNKNOWN value.
- Any read of a register that is in an unimplemented region returns an UNKNOWN value.
- Any write to a register indirected by [SAU\\_RNR.REGION](#) causes all state that is indirected by that register to become UNKNOWN.

*The extension requirements are - S.*

## B8.4 IMPLEMENTATION DEFINED Attribution Unit (IDAU)

R<sub>MVCM</sub>

The IDAU can provide the following Security attribution information for an address:

- Security attribution exempt. This specifies that the address is exempt from security attribution. This information is combined with the address ranges that are architecturally required to be exempt from attribution.
- Non-secure. This specifies if the address is Secure or Non-secure.
- Non-secure callable. This specifies if code at the address can be called from Non-secure state. This attribute is only valid if the address is marked as Secure.
- Region number. This is the region number that matches the address, and is only used by the **TT** instruction.
- Region number valid. This specifies that the region number is valid. This field has no effect on the attribution of the address, and is only used by the **TT** instruction.

*The extension requirements are - S.*

See also:

[TT](#), [TTT](#), [TTA](#), [TTAT](#).

[B8.2 Security attribution on page 218.](#)

## Chapter B9

# The System Timer, SysTick

This chapter specifies the Armv8-M system timer rules. It contains the following section:

[B9.1 \*The system timer, SysTick\* on page 224.](#)

## B9.1 The system timer, SysTick

- R<sub>BORG</sub>** In a PE without the Main Extension and without the Security Extensions, either:
- No system timers are implemented.
  - One system timer, SysTick, is implemented.
- The extension requirements are - !M && !S.*
- R<sub>PDDL</sub>** In a PE without the Main Extension but with the Security Extension, one of the following is true:
- No system timers are implemented.
  - One system timer, SysTick, is implemented. **ICSR.STTNS** determines which Security state owns the SysTick.
  - Two system timers are implemented:
    - SysTick, Secure instance.
    - SysTick, Non-secure instance.
- The extension requirements are - !M && S.*
- R<sub>CNTG</sub>** In a PE with the Main Extension but without the Security Extension, one system timer, SysTick, is implemented.
- The extension requirements are - M && ST && !S.*
- R<sub>XPCW</sub>** In a PE with the Main and Security Extensions, two system timers are implemented:
- SysTick, Secure instance.
  - SysTick, Non-secure instance.
- The extension requirements are - M && S && ST.*
- I<sub>DXSQ</sub>** There are the following SysTick registers:
- SysTick Control and Status Register (**SYST\_CSR**).
  - SysTick Reload Value Register (**SYST\_RVR**).
  - SysTick Current Value Register (**SYST\_CVR**).
  - SysTick Calibration Value Register (**SYST\_CALIB**).
- In a PE with the Security Extension and a SysTick instance dedicated to each Security state, these registers are banked.
- The extension requirements are - ST.*
- I<sub>VHDT</sub>** Each implemented SysTick is a 24-bit decrementing, wrap-on-zero, clear-on-write counter:
- When enabled, the counter counts down from the value in **SYST\_CVR**, **SYST\_CVR**. When it reaches zero, **SYST\_CVR** is reloaded with the value held in **SYST\_RVR** on the next clock edge.
  - Reading **SYST\_CVR** returns the value of the counter at the time of the read access.
  - When the counter reaches zero, it sets **SYST\_CSR.COUNTFLAG** to 1. Reading **SYST\_CSR.COUNTFLAG** clears it to 0.
  - A write to **SYST\_CVR** clears both **SYST\_CVR** and **SYST\_CSR.COUNTFLAG** to 0. **SYST\_CVR** is then reloaded with the value held in **SYST\_RVR** on the next clock edge.
- The extension requirements are - ST.*
- R<sub>TLGK</sub>** Writing the value zero to **SYST\_RVR** disables the SysTick on the next wrap-on-zero. The value zero is held by the counter after the wrap. This is true even when **SYST\_CSR.ENABLE** is 1.
- The extension requirements are - ST.*

Chapter B9. The System Timer, SysTick

B9.1. The system timer, SysTick

- R<sub>TTF</sub>** A write to **SYST\_CVR** does not cause a SysTick exception.  
*The extension requirements are - **ST**.*
- I<sub>VDJQ</sub>** Setting **SYST\_CSR.TICKINT** to 1 causes the SysTick exception to become pending on the SysTick reaching zero.  
*The extension requirements are - **ST**.*
- I<sub>PPGV</sub>** Arm recommends that before enabling a SysTick by **SYST\_CSR.ENABLE**, software writes the required counter value to the **SYST\_RVR**, and then writes to the **SYST\_CVR** to clear the **SYST\_CVR** to zero.  
*The extension requirements are - **ST**.*
- I<sub>MMRQ</sub>** Software can optionally use **SYST\_CALIB.TENMS** to scale the counter to other clock rates within the dynamic range of the counter.  
*The extension requirements are - **ST**.*
- R<sub>QSKV</sub>** When the PE is halted in Debug state, any implemented SysTicks do not decrement.  
*The extension requirements are - **ST** && Halting debug.*
- I<sub>RWFQ</sub>** Each implemented SysTick is clocked by a reference clock, either the PE clock or an external system clock. It is IMPLEMENTATION DEFINED which clock is used as the external reference clock. Arm recommends that if an external system clock is used, the relationship between the PE clock and the external clock is documented, so that system timings can be calculated taking into account metastability, clock skew, and jitter.  
*The extension requirements are - **ST**.*

## Chapter B10

# Nested Vectored Interrupt Controller

This chapter specifies the Armv8-M *Nested Vectored Interrupt Controller* (NVIC) rules. It contains the following sections:

[B10.1 NVIC definition on page 227.](#)

[B10.2 NVIC operation on page 228.](#)

## B10.1 NVIC definition

**R<sub>XJJQ</sub>** An Armv8-M PE includes an integral interrupt controller.

**R<sub>QHG</sub>** The Interrupt Controller Type Register (**ICTR**) defines the number of external interrupt lines that are supported.

See also:

*[ICTR, Interrupt Controller Type Register.](#)*

## B10.2 NVIC operation

- R<sub>SNVK</sub>** It is IMPLEMENTATION DEFINED which NVIC interrupts are implemented.
- R<sub>SGCR</sub>** When a particular NVIC interrupt line is not implemented, the registers that are associated with it are reserved.
- R<sub>CCVJ</sub>** Only an interrupt that is both pending and enabled with sufficient priority can preempt PE execution.
- R<sub>CVJS</sub>** The following events on the input associated with an interrupt cause the pending state associated with the interrupt to become set:
- The input is HIGH while the active state associated with the interrupt is clear.
  - The input transitions from LOW to HIGH while the active state associated with the interrupt is set.
- I<sub>WTFS</sub>** The Armv8-M interrupt behavior provides compatibility with both active-high level-sensitive and pulse-sensitive interrupt signaling:
- For level-sensitive interrupts, the associated exception handler runs one time for each occurrence as long as the level is cleared before the exception handler returns. If the level of the input is HIGH after the exception handler returns, the exception will be pended again.
  - For pulse-sensitive interrupts, the associated exception handler runs one time only, regardless of the number of pulses that the NVIC sees before the exception handler is entered. If a pulse occurs after the exception handler has been entered, the exception will be pended again.
- I<sub>HVQQ</sub>** For some implementations, pulse-sensitive interrupt signals are held long enough to ensure that the PE can sample them reliably.
- R<sub>QKFW</sub>** All NVIC interrupts have a programmable priority value and an associated exception number.
- R<sub>XNQW</sub>** NVIC interrupts can be enabled and disabled by writing to their corresponding Interrupt Set-Enable or Interrupt Clear-Enable register bit field.
- R<sub>WGDJ</sub>** An implementation can hard-wire interrupt enable bits to zero if the associated interrupt line does not exist.
- R<sub>RSDJ</sub>** An implementation can hard-wire interrupt enable bits to one if the associated interrupt line cannot be disabled.
- R<sub>NRJV</sub>** It is IMPLEMENTATION DEFINED for each NVIC interrupt line supported whether an NVIC interrupt supports either or both setting and clearing of the associated pending state under software control.

See also:

[B3.9 Exception numbers and exception priority numbers on page 65.](#)



[B3.13 Priority model on page 77.](#)

[Nested Vectored Interrupt Controller Block.](#)

[Nested Vectored Interrupt Controller Block\(NS alias\).](#)

## Chapter B11

### Debug

This chapter specifies the Armv8-M debug rules. It contains the following sections:

- [B11.1 \*Debug feature overview\* on page 231.](#)
- [B11.2 \*Accessing debug features\* on page 236.](#)
- [B11.3 \*Debug authentication interface\* on page 240.](#)
- [B11.4 \*Debug event behavior\* on page 250.](#)
- [B11.5 \*Debug state\* on page 263.](#)
- [B11.6 \*Exiting Debug state\* on page 266.](#)
- [B11.7 \*Multiprocessor support\* on page 267.](#)

## B11.1 Debug feature overview

**R<sub>WXRJ</sub>** The debug configuration of an implementation is IMPLEMENTATION DEFINED.

**R<sub>FMQF</sub>** The following table describes the optional features of the Armv8-M debug architecture.

| Feature                                      | Main Extension                       | Baseline Implementation       |
|----------------------------------------------|--------------------------------------|-------------------------------|
| <b>DebugMonitor exception</b>                | Always implemented                   | Never implemented             |
| <b>Halting debug</b>                         | Optional                             | Optional                      |
| <b>EDBGRQ</b><br><i>External Halt signal</i> | Optional                             | Requires Halting debug        |
| <b>Flash Patch and Breakpoint unit - FPB</b> | Optional                             | Requires Halting debug        |
| <b>Data Watchpoint and Trace Unit - DWT</b>  |                                      |                               |
| <i>Debug functionality - DWT-D</i>           | Optional                             | Requires Halting debug        |
| <i>Trace functionality - DWT-T</i>           | Requires ITM and Debug functionality | Never implemented             |
| <b>Instrumentation Trace Macrocell - ITM</b> | Optional                             | Never implemented             |
| <b>Cross Trigger Interface - CTI</b>         | Requires ETM or Halting Debug        | Requires ETM or Halting Debug |
| <b>Trace Port Interface Unit - TPIU</b>      | Requires ITM or ETM                  | Requires ETM                  |
| <b>Embedded Trace Macrocell - ETM</b>        | Optional                             | Optional                      |

**R<sub>FHRN</sub>** The following optional debug components are not part of the Armv8-M architecture:

- The *Cross-Trigger Interface (CTI)*.
- The CoreSight basic trace router (MTB).
- The Embedded Trace Macrocell (ETM).

*Note, CTI requires Halting debug or ETM.*

**I<sub>SFSG</sub>** The recommended Debug implementation levels are:

- Minimum.
- Basic.
- Comprehensive.
- Program trace.

### Minimum

In an implementation that includes the Main Extension, the minimum level contains support for the DebugMonitor exception, including:

- The **BKPT** instruction.
- **DEMCR** Monitor debug features.
- Monitor entry from External debug requests.
- **DFSR**.

**DHCSR**, **DCRSR**, **DCRDR**, and the Halting debug features in **DFSR** and **DEMCR** are RES0. **ID\_DFR0** is RAZ.

In an implementation that does not include the Main Extension there is no debug support.

**DFSR**, **DHCSR**, **DCRSR**, **DCRDR**, and **DEMCR** are RES0. **ID\_DFR0** is RAZ.

### Basic

In an implementation that includes the Main Extension, the basic level adds support for Halting debug with:

- A Debug Access Port and ROM table.
- [DHCSR](#), [DCRSR](#), [DCRDR](#), and the Halting debug features in [DEMCR](#) are implemented.
- FPB with at least two breakpoints.
- DWT with at least:
  - One watchpoint that supports instruction, data address, and data value matching.
  - [DWT\\_PCSR](#).
- Optional support for a CTI in a multiprocessor system.

Support for the basic implementation is identified by [ID\\_DFR0](#).

In an implementation that does not include the Main Extension, the basic level adds support for Halting debug with:

- A Debug Access Port and ROM table.
- [SHCSR](#), [DFSR](#), [DHCSR](#), [DCRSR](#), [DCRDR](#), and [DEMCR](#) are implemented. Access for the PE is [IMPDEF](#).
- FPB with at least two breakpoints.
- DWT with at least:
  - One watchpoint that supports instruction, data address, and data value matching.
  - [DWT\\_PCSR](#).
- Optional support for a CTI in a multiprocessor system.

Support for the basic implementation is identified by [ID\\_DFR0](#).

### Comprehensive

In an implementation that includes the Main Extension, the comprehensive level adds basic trace support with:

- An ITM.
- DWT with:
  - Trace support.
  - Profiling support.
  - Cycle counter.
- TPIU.

In an implementation that does not include the Main Extension, there is no support for the comprehensive level.

### Program trace

In an implementation that includes the Main Extension, Program trace adds support for ETMs.

In an implementation that does not include the Main Extension, Program trace adds supports for ETMs and TPIUs.

The extension requirements are - [DB](#).

See also:

[B11.1.1 Debug mechanisms on page 233](#).

Halting debug.  
DebugMonitor exception.  
B11.4.4 *Breakpoint instructions* on page 261.  
B12.1 *Instrumentation Trace Macrocell* on page 269.  
B12.2 *Data Watchpoint and Trace unit* on page 278.  
B12.3 *Embedded Trace Macrocell* on page 298.  
B12.4 *Trace Port Interface Unit* on page 299.  
B12.5 *Flash Patch and Breakpoint unit* on page 301.  
*DEMCR, Debug Exception and Monitor Control Register.*  
*DFSR, Debug Fault Status Register.*  
*DHCSR, Debug Halting Control and Status Register.*  
*DCRDR, Debug Core Register Data Register.*  
*DCRSR, Debug Core Register Select Register.*  
*ID\_DFR0, Debug Feature Register.*  
*DWT\_PCSR, DWT Program Control Sample Register .*

### B11.1.1 Debug mechanisms

**R<sub>HWCH</sub>** Armv8-M supports a range of invasive and non-invasive debug mechanisms. The *invasive debug mechanisms* are:

- The ability to halt the PE. This provides a run-stop debug model.
- Debugging code using the DebugMonitor exception. This provides less intrusive debug than halting the PE.

The *non-invasive debug techniques* are:

- Generating application trace by writing to the *Instrumentation Trace Macrocell* (ITM), causing a low level of intrusion.
- Non-intrusive program trace and profiling.

The extension requirements are - **DB**. Note, *M* is required for the DebugMonitor exception and ITM.

**I<sub>LBLE</sub>** When the PE is halted, it is in *Debug state*.  
The extension requirements are - **Halting debug**.

**I<sub>SXVR</sub>** When the PE is not halted, it is in *Non-debug state*.  
The extension requirements are - **Halting debug**.

See also:

[B11.2 Accessing debug features](#) on page 236.

### B11.1.2 Debug resources

**R<sub>TZVG</sub>** In the system address map, debug resources are in the *Private Peripheral Bus* (PPB) region.

**R<sub>FBHD</sub>** Except for the resources in the SCS, each debug component occupies a fixed 4KB address region.

Chapter B11. Debug  
B11.1. Debug feature overview

The extension requirements are - **DB**.

- R<sub>WXTK</sub>** The debug resources in the SCS are:
- The *Debug Control Block* (DCB).
  - Debug controls in the *System Control Block* (SCB).

The extension requirements are - **DB**.

- I<sub>KKBT</sub>** If the Main Extension is implemented, then support for DebugMonitor is implemented. If the Main Extension is not implemented, then DebugMonitor is not supported.

The extension requirements are - **DB**. Note, *M* is required for DebugMonitor exception.

- R<sub>VMGD</sub>** ROM table entries identify which optional debug components are implemented.

The extension requirements are - **DB**.

- R<sub>RNXK</sub>** The addresses of the optional debug resources are:

| Address range          | Debug Resource                                                                                        |                                                  |
|------------------------|-------------------------------------------------------------------------------------------------------|--------------------------------------------------|
| 0xE0000000-0xE0000FFF  | Instrumentation Trace Macrocell (ITM)                                                                 |                                                  |
| 0xE0001000-0xE0001FFF  | Data Watchpoint and Trace (DWT) Unit                                                                  |                                                  |
| 0xE0002000-0xE0002FFF  | Flashpatch and Breakpoint Unit (FPB)                                                                  |                                                  |
| 0xE000E000-0xE000EFFF  | Secure SCS                                                                                            |                                                  |
|                        | 0xE000ED00-0xE000ED8F                                                                                 | Secure and Non-secure System Control Block (SCB) |
|                        | 0xE000EDF0-0xE000EFFF                                                                                 | Secure and Non-secure Debug Control Block (DCB)  |
| 0xE002E000-0xE002EFFF  | Non-secure SCS                                                                                        |                                                  |
|                        | 0xE002ED00-0xE002ED8F                                                                                 | Non-secure System Control Block (SCB)            |
|                        | 0xE002EDF0-0xE002EFFF                                                                                 | Non-secure Debug Control Block (DCB)             |
| 0xE0040000-0xE0040FFF  | Trace Port Interface Unit (TPIU),<br>when not implemented as a<br>shared resource otherwise reserved. |                                                  |
| 0xE0041000-0xE0041FFF  | Embedded Trace Macrocell (ETM)                                                                        |                                                  |
| 0xE0042000-0xE00FEFFF- | IMPLEMENTATION DEFINED                                                                                |                                                  |
| 0xE00FF000-0xE00FFFFF  | ROM table                                                                                             |                                                  |

The extension requirements are - **DB**.

See also:

- [B12.1 Instrumentation Trace Macrocell on page 269.](#)
- [B12.2 Data Watchpoint and Trace unit on page 278.](#)
- [B12.5 Flash Patch and Breakpoint unit on page 301.](#)
- [Chapter B6 The System Address Map on page 198.](#)
- [B11.2.2 Debug System registers on page 238.](#)
- [B12.4 Trace Port Interface Unit on page 299.](#)
- [B12.3 Embedded Trace Macrocell on page 298.](#)
- [B11.2.1 ROM table on page 236.](#)
- [B11.2 Accessing debug features on page 236.](#)

### B11.1.3 Trace

- R<sub>LJVL</sub>** Trace can be generated by using the:
- *Embedded Trace Macrocell* (ETM).

Chapter B11. Debug

B11.1. Debug feature overview

- *Instrumentation Trace Macrocell (ITM).*
- *Data Watchpoint and Trace (DWT) unit.*

*The extension requirements are - ETM || ITM || DWT-T. Note, ITM requires M.*

**R<sub>NFVB</sub>**

A debug implementation that generates trace includes a trace sink, such as a TPIU.

*The extension requirements are - (ETM || ITM || DWT-T) && TPIU. Note, ITM requires M.*

**I<sub>RJKJ</sub>**

A TPIU can be either the Armv8-M TPIU implementation, or an external system resource.

*The extension requirements are - ETM || ITM || DWT-T.*

See also:

[ITM and DWT Packet Protocol Specification.](#)

The applicable ETM Architecture Specification.

## B11.2 Accessing debug features

- R<sub>WVSZ</sub>** The mechanism by which an external debugger accesses the PE and system is IMPLEMENTATION DEFINED.  
*The extension requirements are - [DB](#).*
- I<sub>QPHR</sub>** A debugger can use a *Debug Access Port* (DAP) interface, such as the DAP described in the *Arm®Debug Interface v5 Architecture Specification*(ADIV5), to interrogate a system for Memory Access Ports (MEM-APs). The base register in a Memory Access Port provides the address of the ROM table, or the first in a series of ROM tables in a ROM table hierarchy. The Memory Access Port can then fetch the ROM table entries. Arm recommends implementation of an ADIV5 DAP for compatibility with tools.  
*The extension requirements are - [DB](#).*
- R<sub>WPGQ</sub>** Writes from a DAP are complete when the DAP reports the writes as complete.  
*The extension requirements are - [DB](#).*
- R<sub>WCQK</sub>** For SCS registers, a write from a DAP is complete when the write has completed and the SCS register has been updated.  
*The extension requirements are - [DB](#).*
- R<sub>JRHS</sub>** Software configures and controls the debug model through memory-mapped registers.  
*The extension requirements are - [DB](#).*
- R<sub>KSJT</sub>** The Debug Access Port is an observer, and observes all accesses that the PE makes.  
*The extension requirements are - [DB](#).*

See also:

[B11.2.1 ROM table](#) .

[B11.3.4 DAP access permissions](#) on page 245.

The *Arm®Debug Interface v5 Architecture Specification*.

### B11.2.1 ROM table

- I<sub>XFVN</sub>** The ROM table is a table of entries providing a mechanism to identify the debug infrastructure that is supported by the implementation.  
*The extension requirements are - [DB](#).*
- I<sub>FWPG</sub>** The ROM table indicates the implemented debug components, and the position of those components in the memory map. See the *Arm®Debug Interface v5 Architecture Specification* for the format of a ROM table entry.  
*The extension requirements are - [DB](#).*
- I<sub>PHJJ</sub>** For an Armv8-M ROM table, all entry offsets are negative. The ROM table entry points to the top of a 4KB page, the offset points to the bottom of that page that contains the Peripheral and Component ID registers.  
*The extension requirements are - [DB](#).*
- R<sub>GPPX</sub>** The ROM table is implemented if any other debug component is implemented or a Debug Access Port is implemented.  
*The extension requirements are - [DB](#).*



Chapter B11. Debug

B11.2. Accessing debug features

**R<sub>BQSP</sub>** Bit[0] of the ROM table entries indicates whether the corresponding debug component is implemented and is accessible through the PPB at the indicated address. If the corresponding debug component is not implemented, this bit has a value of 0.

The extension requirements are - **DB**.

**R<sub>NDQW</sub>** If a debug component is implemented, debug registers can provide additional information about the implemented features of that debug component.

The extension requirements are - **DB**.

**R<sub>DPVG</sub>** The format of the ROM table is:

| Offset           | Value                       | Name     | Description                                                                                                                                                                                       |
|------------------|-----------------------------|----------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 0x000            | 0xFFF0F003                  | ROMSCS   | Points to the SCS at 0xE000E000                                                                                                                                                                   |
| 0x004            | 0xFFF02002 or<br>0xFFF02003 | ROMDWT   | Points to the Data Watchpoint and Trace unit at 0xE0001000                                                                                                                                        |
| 0x008            | 0xFFF03002 or<br>0xFFF03003 | ROMFPB   | Points to the Flash Patch and Breakpoint unit at 0xE0002000                                                                                                                                       |
| 0x00C            | 0xFFF01002 or<br>0xFFF01003 | ROMITM   | Points to Instrumentation Trace unit at 0xE0000000.                                                                                                                                               |
| 0x010            | 0xFFF41002 or<br>0xFFF41003 | ROMTPIU  | Points to the Trace Port Interface Unit.                                                                                                                                                          |
| 0x014            | 0xFFF42002 or<br>0xFFF42003 | ROMETM   | Points to the Embedded Trace Macrocell.                                                                                                                                                           |
| -                | 0x00000000                  | End      | End of table marker. It is IMPDEF whether the table is extended with pointers to other system debug resources. The table entries always terminate with a null entry.                              |
| 0x020 -<br>0xEFC | -                           | Not used | Reserved for additional ROM table entries.                                                                                                                                                        |
| 0xF00 -<br>0xFC8 | -                           | Reserved | Reserved, not used for ROM table entries.                                                                                                                                                         |
| 0xFCC            | 0x00000001                  | MEMTYPE  | Bit [0] is set to 1 to indicate that resources other than those listed in the ROM table are accessible in the same 32-bit address space, using the DAP.Bits [31:1] of the MEMTYPE entry are RES0. |
| 0xFD0            | IMP DEF                     | PIDR4    | CIDRx values are fully defined for the ROM table, and are CorseSight compliant. PIDRx values are CoreSight compliant or RAZ.                                                                      |
| 0xFD4            | 0                           | PIDR5    |                                                                                                                                                                                                   |
| 0xFD8            | 0                           | PIDR6    |                                                                                                                                                                                                   |
| 0xFDC            | 0                           | PIDR7    |                                                                                                                                                                                                   |
| 0xFE0            | IMP DEF                     | PIDR0    |                                                                                                                                                                                                   |
| 0xFE4            | IMP DEF                     | PIDR1    |                                                                                                                                                                                                   |
| 0xFE8            | IMP DEF                     | PIDR2    |                                                                                                                                                                                                   |
| 0xFEC            | IMP DEF                     | PIDR3    |                                                                                                                                                                                                   |
| 0xFF0            | 0x0000000D                  | CIDR0    |                                                                                                                                                                                                   |
| 0xFF4            | 0x00000010                  | CIDR1    |                                                                                                                                                                                                   |
| 0xFF8            | 0x00000005                  | CIDR2    |                                                                                                                                                                                                   |
| 0xFFC            | 0x000000B1                  | CIDR3    |                                                                                                                                                                                                   |

Accesses to the ROMITM cannot cause a non-existent memory exception.

It is IMPLEMENTATION DEFINED whether the ETM and TPIU are a shared resource and whether the resource is managed by the local PE or a different resource.

The extension requirements are - **DB** and those indicated in the table.

**R<sub>RGVM</sub>** The entry 0x00000000 is the end-of-table marker.

The extension requirements are - [DB](#).

See also:

[B11.2.3 CoreSight and identification registers](#) .

## B11.2.2 Debug System registers

**R<sub>RHDW</sub>** The debug provision in the *System Control Block* (SCB) comprises:

- Two handler-related flag bits, [ICSR.ISRPREEMPT](#) and [ICSR.ISRPENDING](#).
- The [DFSR](#).

The extension requirements are - [DB](#).

See also:

[Chapter D1, Register Specification](#).

[Debug Control Block](#).

## B11.2.3 CoreSight and identification registers

**I<sub>CMLH</sub>** Arm recommends that CoreSight-compliant ID registers are implemented to allow identification and discovery of the components to a debugger.

The extension requirements are - [DB](#).

**R<sub>CBCM</sub>** The address spaces that are reserved in each of the debug components for IMPLEMENTATION DEFINED ID registers and CoreSight compliance are:

| Debug Component | Space reserved for ID registers | Space reserved for CoreSight compliance |
|-----------------|---------------------------------|-----------------------------------------|
| ITM             | 0xE0000FD0-0xE0000FFC           | 0xE0000FA0-0xE0000FCC                   |
| DWT             | 0xE0001FD0-0xE0001FFC           | 0xE0001FA0-0xE0001FCC                   |
| FPB             | 0xE0002FD0-0xE0002FFC           | 0xE0002FA0-0xE0002FCC                   |
| SCS             | 0xE000EFD0-0xE000EFFC           | 0xE000EFA0-0xE000EFCC                   |
| TPIU            | 0xE0040FD0-0xE0040FFC           | 0xE0040FA0-0xE0040FCC                   |
| ETM             | 0xE0041FD0-0xE0041FFC           | 0xE0041FA0-0xE0041FCC                   |
| ROM table       | 0xE00FFFD0-0xE00FFFFC           | 0xE00FFFA0-0xE00FFFCC                   |

The extension requirements are - [DB](#).

**R<sub>VWSX</sub>** For the ROM table, the ID register space is used for a set of CoreSight-compliant ID registers.

The extension requirements are - [DB](#).

**R<sub>HDXK</sub>** For all components other than the ROM table, if the registers in the ID register space are not used for ID registers they are RAZ.

The extension requirements are - [DB](#).

**R<sub>VQPM</sub>** If CoreSight-compliant ID registers are implemented, the Class field in Component ID Register 1 is:

- 0x1 for the ROM table.
- 0x9 for other components.

The extension requirements are - [DB](#).

- $I_{HQSR}$  The Part number in the PIDR registers must be assigned a unique value for each implementation, or Unique Component Identifier, as with all other CoreSight components.
- CoreSight permits that two or more functionally different components are permitted to share the same Part number, so long as they have different values of the [DDEVTYPE](#) or [DDEVARCH](#) registers.
- The extension requirements are - **DB**.*
- $I_{CTBF}$  The Part number in the PIDR registers do not need to be unique for different implementation options of the same part.
- The extension requirements are - **DB**.*

## B11.3 Debug authentication interface

**I<sub>GWTN</sub>** The following pseudocode functions provide an abstracted description of the authentication interface:

- `ExternalInvasiveDebugEnabled()`.
- `ExternalSecureInvasiveDebugEnabled()`.
- `ExternalNoninvasiveDebugEnabled()`.
- `ExternalSecureNoninvasiveDebugEnabled()`.

The extension requirements are - **DB**.

**R<sub>SWWT</sub>** For an implementation using the CoreSight signals **DBGEN**, **NIDEN**, **SPIDEN**, and **SPNIDEN**:

- `ExternalInvasiveDebugEnabled()` returns TRUE if **DBGEN** is asserted.
- `ExternalSecureInvasiveDebugEnabled()` returns TRUE if both **DBGEN** and **SPIDEN** are asserted.
- `ExternalNoninvasiveDebugEnabled()` returns TRUE if either **NIDEN** or **DBGEN** is asserted.
- `ExternalSecureNoninvasiveDebugEnabled()` returns TRUE if both of the following conditions apply:
  - Either **NIDEN** or **DBGEN** is asserted.
  - Either **SPNIDEN** or **SPIDEN** is asserted.

The extension requirements are - **DB**.

**R<sub>HVGN</sub>** For any implementation of the authentication interface, if `ExternalInvasiveDebugEnabled()` is FALSE, then `ExternalSecureInvasiveDebugEnabled()` is FALSE.

The extension requirements are - **DB**.

**R<sub>JWCS</sub>** For any implementation of the authentication interface, if `ExternalNoninvasiveDebugEnabled()` is FALSE, then `ExternalSecureNoninvasiveDebugEnabled()` is FALSE.

The extension requirements are - **DB**.

**R<sub>XCMD</sub>** For any implementation of the authentication interface, if `ExternalInvasiveDebugEnabled()` is TRUE, then `ExternalNoninvasiveDebugEnabled()` is TRUE.

The extension requirements are - **DB**.

**R<sub>LCHH</sub>** For any implementation of the authentication interface, if `ExternalSecureInvasiveDebugEnabled()` is TRUE, then `ExternalSecureNoninvasiveDebugEnabled()` is TRUE.

The extension requirements are - **DB**.

**I<sub>MSRG</sub>** Secure self-hosted debug is controlled by the authentication interface. The pseudocode function `ExternalSecureSelfHostedDebugEnabled()` provides an abstracted description of this authentication interface.

The extension requirements are - **DB**.

**R<sub>GLWM</sub>** Between a change to the debug authentication interface and a following [Context synchronization event](#), it is UNPREDICTABLE whether the PE uses the old or the new values.

The extension requirements are - **DB**.

See also:

[B11.3.1 Halting debug authentication on page 241](#).

[B11.3.3 DebugMonitor exception authentication on page 243.](#)

[B11.3.2 Non-invasive debug authentication on page 242.](#)

[B11.3.4 DAP access permissions on page 245.](#)

### B11.3.1 Halting debug authentication

**I<sub>DMFG</sub>** Halting debug authentication is controlled by the IMPLEMENTATION DEFINED authentication interface function [ExternalInvasiveDebugEnabled\(\)](#), and if the Security Extension is implemented, the IMPLEMENTATION DEFINED authentication interface function [ExternalSecureInvasiveDebugEnabled\(\)](#).

*The extension requirements are - **Halting debug**. Note, External Secure invasive debug requires S.*

**R<sub>JJK</sub>** Unless otherwise stated Halting is prohibited in all states if the function [ExternalInvasiveDebugEnabled\(\)](#) returns FALSE.

*The extension requirements are - **Halting debug**.*

**R<sub>JXTX</sub>** When the PE is halted, the PE behaves as if [ExternalInvasiveDebugEnabled\(\)](#) is TRUE. The pseudocode function [HaltingDebugAllowed\(\)](#) describes this.

*The extension requirements are - **Halting debug**.*

**I<sub>BCZM</sub>** If the Security Extension is not implemented, there are two Halting debug authentication modes:

| <a href="#">ExternalInvasiveDebugEnabled()</a> | <b>DHCSR.S_HALT</b> | <b>Halting debug authentication mode</b> |
|------------------------------------------------|---------------------|------------------------------------------|
| FALSE                                          | 0                   | Halting is prohibited.                   |
| FALSE                                          | 1                   | Halting is allowed.                      |
| TRUE                                           | X                   | Halting is allowed.                      |

*The extension requirements are - **Halting debug && !S**.*

**R<sub>BMRJ</sub>** Halting is prohibited in Secure state if any of:

- [ExternalInvasiveDebugEnabled\(\)](#) returns FALSE.
- [DAUTHCTRL.SPIDENSEL](#) is set to 1 and [DAUTHCTRL\\_S.INTSPIDEN](#) is set to 0.
- [DAUTHCTRL.SPIDENSEL](#) is set to 0 and [ExternalSecureInvasiveDebugEnabled\(\)](#) returns FALSE.

The pseudocode function [SecureHaltingDebugAllowed](#) describes this.

*The extension requirements are - **Halting debug && S**.*

**R<sub>QTBK</sub>** Halting is prohibited in unprivileged modes in Secure state if all of:

- [SecureHaltingDebugAllowed\(\)](#) returns FALSE.

*The extension requirements are - **Halting debug && S**.*

**R<sub>KBKM</sub>** If the PE is in non-Debug state the following condition is true:

- [DHCSR.S\\_SDE](#) reads as one if any one of the following of true, and reads as zero otherwise:
  - [SecureHaltingDebugAllowed\(\)](#) returns TRUE.

*The extension requirements are - **Halting debug**. Note, S is required for Secure Behavior.*

**R<sub>KMXG</sub>** If the PE is in Debug state:

- [DHCSR.S\\_SDE](#) reads as one if either of the the following is true, and reads as zero otherwise:

- The PE entered Debug state from Secure state.
- The PE entered Debug state from Non-secure state when `SecureHaltingDebugAllowed()` returned TRUE.

The extension requirements are - **Halting debug**. Note, *S* is required for Secure behavior.

**I<sub>LDTR</sub>** If the Security Extension is implemented, there are three Halting debug authentication modes:

| HaltingDebugAllowed() | <b>DHCSR.S_SDE</b> | <b>Halting debug authentication mode</b>                                          |
|-----------------------|--------------------|-----------------------------------------------------------------------------------|
| FALSE                 | X                  | Halting is prohibited.                                                            |
| TRUE                  | 0                  | Halting is allowed in Non-secure state.<br>Halting is prohibited in Secure state. |
|                       | 1                  | Halting is allowed.                                                               |

The extension requirements are - **Halting debug && S**.

**R<sub>FXCB</sub>** When `DHCSR.C_DEBUGEN == 0` or the PE is in a state in which Halting is prohibited, the PE does not enter Debug state.

The extension requirements are - **Halting debug**. Note, *S* is required for Secure behavior.

See also:

`CanHaltOnEvent()`.

### B11.3.2 Non-invasive debug authentication

**R<sub>GFTG</sub>** Non-invasive authentication is controlled by the IMPLEMENTATION DEFINED function:

`ExternalNoninvasiveDebugEnabled()`.

The extension requirements are - **DB**.

**R<sub>HXQD</sub>** Secure Non-invasive authentication is controlled by the IMPLEMENTATION DEFINED function:

`ExternalSecureNoninvasiveDebugEnabled()`.

The extension requirements are - **DB**.

**R<sub>CFNB</sub>** When `HaltingDebugAllowed()` is TRUE:

- The PE behaves as if `ExternalNoninvasiveDebugEnabled()` returns TRUE.
- The pseudocode function `NoninvasiveDebugAllowed()` describes this.

The extension requirements are - **DB**.

**R<sub>QMRF</sub>** Non-invasive debug is prohibited if the functions `SecureNoninvasiveDebugAllowed()` and `NoninvasiveDebugAllowed()` both return FALSE.

The extension requirements are - **DB**.

**I<sub>PHPR</sub>** If the Security Extension is not implemented, there are two non-invasive debug authentication modes:

| ExternalNon-invasiveDebugEnabled() | HaltingDebugAllowed() | <b>Non-invasive debug authentication mode</b> |
|------------------------------------|-----------------------|-----------------------------------------------|
| FALSE                              | FALSE                 | Non-invasive debug prohibited.                |
|                                    | TRUE                  | Non-invasive debug allowed.                   |
| TRUE                               | X                     | Non-invasive debug allowed.                   |

The extension requirements are - **DB**.

- I<sub>MLPS</sub>** Non-invasive debug of Secure operations is prohibited if any of the following are true:
- `NoninvasiveDebugEnabled()` returns FALSE.
  - `DHCSR.S_SDE` is set to 0, `DAUTHCTRL.SPIDENSEL` is set to 1, and `DAUTHCTRL.INTSPIDEN` is set to 0.
  - `ExternalNoninvasiveDebugEnabled()` returns FALSE, `DHCSR.S_SDE` is set to 0, and `DAUTHCTRL.SPIDENSEL` is set to 0.

The pseudocode function `SecureNoninvasiveDebugEnabled()` shows this, if this function returns true Secure Non-invasive debug is permitted.

The extension requirements are - **DB**.

- I<sub>PNRC</sub>** If the Security Extension is implemented, there are three non-invasive debug authentication modes:

| Noninvasive-<br>DebugEnabled() | SecureNon-<br>invasiveDebugEnabled() | Non-invasive debug<br>authentication mode                                                                        |
|--------------------------------|--------------------------------------|------------------------------------------------------------------------------------------------------------------|
| FALSE                          | X                                    | Non-invasive debug prohibited.                                                                                   |
| TRUE                           | FALSE                                | Non-invasive debug of only Non-secure operations allowed.<br>Non-invasive debug of Secure operations prohibited. |
|                                | TRUE                                 | Non-invasive debug allowed.                                                                                      |

The extension requirements are - **DB && S**.

- R<sub>LXRK</sub>** The PE does not generate any trace or profiling data when non-invasive debug is prohibited.

The extension requirements are - **DB**.

- R<sub>VYGT</sub>** If non-invasive debug of Secure operations is prohibited, the PE does not generate any trace or profiling data that contains secure information or is attributable to secure operations.

The extension requirements are - **DB && S**.

- R<sub>TWDH</sub>** If non-invasive debug is prohibited in the current Security state, an ETM behaves as described in the relevant ETM architecture.

The extension requirements are - **DB && S && ETM**.

See also:

`NoninvasiveDebugEnabled()`.

`SecureNoninvasiveDebugEnabled()`.

[B12.2.2 DWT unit operation on page 279](#).

### B11.3.3 DebugMonitor exception authentication

- R<sub>MXTM</sub>** DebugMonitor exception authentication is only available if the Main Extension is implemented.

The extension requirements are - **M**.

- R<sub>LQCN</sub>** DebugMonitor exception authentication is controlled by the IMPLEMENTATION DEFINED authentication interface function `ExternalSecureSelfHostedDebugEnabled()`.

The extension requirements are - **M && S**.

- R<sub>QJQN</sub>** Unless otherwise stated DebugMonitor exceptions are never generated for Secure operations if any of:

- `DAUTHCTRL.SPIDENSEL` is set to 1 and `DAUTHCTRL.INTSPIDEN` is set to 0.
- `DAUTHCTRL.SPIDENSEL` is set to 0 and `ExternalSecureSelfHostedDebugEnabled()` returns FALSE.

The pseudocode function `SecureDebugMonitorAllowed()` describes this.

The extension requirements are - **M** && **S**.



- R<sub>CPPN</sub>** When a DebugMonitor exception is pending or active:
- **DEMCR.SDME** is set to 1 if `SecureDebugMonitorAllowed()` returned TRUE when a DebugMonitor exception became pending or active.
  - **DEMCR.SDME** is zero otherwise.

The extension requirements are - **M**.

- R<sub>WXMG</sub>** When a DebugMonitor exception is not pending and is not active:
- **DEMCR.SDME** is set to 1 if `SecureDebugMonitorAllowed()` is TRUE.
  - **DEMCR.SDME** is zero otherwise.

The extension requirements are - **M**.

- I<sub>RZXJ</sub>** If the Security Extension is implemented, there are two DebugMonitor exception authentication modes, which are controlled by **DEMCR.SDME**:

| <b>DEMCR.SDME</b> | <b>Target State for DebugMonitor exception</b> | <b>DebugMonitor exception authentication mode</b> |
|-------------------|------------------------------------------------|---------------------------------------------------|
| 0                 | Non-secure                                     | Non-secure DebugMonitor exception.                |
| 1                 | Secure                                         | Secure DebugMonitor exception.                    |

The extension requirements are - **M** && **S**.

- R<sub>YFPK</sub>** If **DEMCR.SDME** == 1, **SHPR3.PRI\_12** behaves as RAZ/WI when accessed from Non-secure state.

The extension requirements are - **M** && **S**.

- R<sub>HXLX</sub>** When set to 1, **DEMCR.MON\_PEND** remains set to 1 until either the DebugMonitor exception is taken or a write sets the field to 0.

The extension requirements are - **M**.

See also:

`CanPendMonitorOnEvent()`.

### B11.3.4 DAP access permissions

- R<sub>BFSB</sub>** When `HaltingDebugAllowed()` returns TRUE the external debugger can issue requests to the entire physical address space.

The extension requirements are - **DB**.

- R<sub>DVSN</sub>** Unless otherwise stated, if `HaltingDebugAllowed() = FALSE` the DAP access permissions are:

| Address Range           | Region or registers                 | NoninvasiveDebugAllowed()         |           |
|-------------------------|-------------------------------------|-----------------------------------|-----------|
|                         |                                     | FALSE                             | TRUE      |
| 0x00000000-0xDFFFFFFF   | Rest of Memory                      | No access                         | No access |
| 0xE0000000-0xE00FFFFFFF | PPB                                 |                                   |           |
|                         | 0xE00xxFB0-0xE00xxFB7               | CoreSight Software Lock registers | No access |
|                         | 0xE00xxFD0-0xE00xxFFF               | All ID registers                  | RO        |
|                         | 0xE0000000-0xE0000FCF               | ITM                               | No access |
|                         | 0xE0001000-0xE0001FCF               | DWT                               | No access |
|                         | 0xE0040000-0xE0040FFF               | TPIU                              | RW        |
|                         | 0xE0041000-0xE0041FFF               | ETM                               | RW        |
|                         | 0xE0042000-0xE00FEFFF               | IMPDEF                            | IMPDEF    |
|                         | 0xE00FF000-0xE00FFFFFFF             | ROM table                         | RO        |
|                         | All other PPB regions and registers | No access                         | No access |
| 0xE0100000-0xFFFFFFFF   | Vendor_SYS                          | No access                         | RW        |

0xE00xxFB0-0xE00xxFB7 is for each debug component implementing the CoreSight Software Lock registers. These registers are optional. 0xE00xxFD0-0xE00xxFFF for each debug component implementing the CoreSight ID registers. These registers are optional.

The extension requirements are - [DB](#).

**R<sub>FFPN</sub>** The DAP is capable of requesting Secure and Non-secure accesses.

The extension requirements are - [DB](#) && [S](#).

**I<sub>PQSV</sub>** The architecture does not describe how a DAP requests Secure or Non-secure memory accesses. In the recommended ADIV5 Memory Access Port (MEM-AP), Arm recommends that:

- CSW[30], CSW.Prot[6], selects a Secure or Non-secure access:
  - **0**: Request a Secure access.
  - **1**: Request a Non-secure access.
- CSW[23], CSW.SPIDEN, is Read-As-One. This is because the DAP can always request a Secure access.

The extension requirements are - [DB](#) && [S](#).

**I<sub>DPHD</sub>** In a CoreSight DAP, the **SPIDEN** input to the Armv8-M MEM-AP is independent of the SPIDEN input of the PE, and must be tied HIGH.

The extension requirements are - [DB](#) && [S](#).

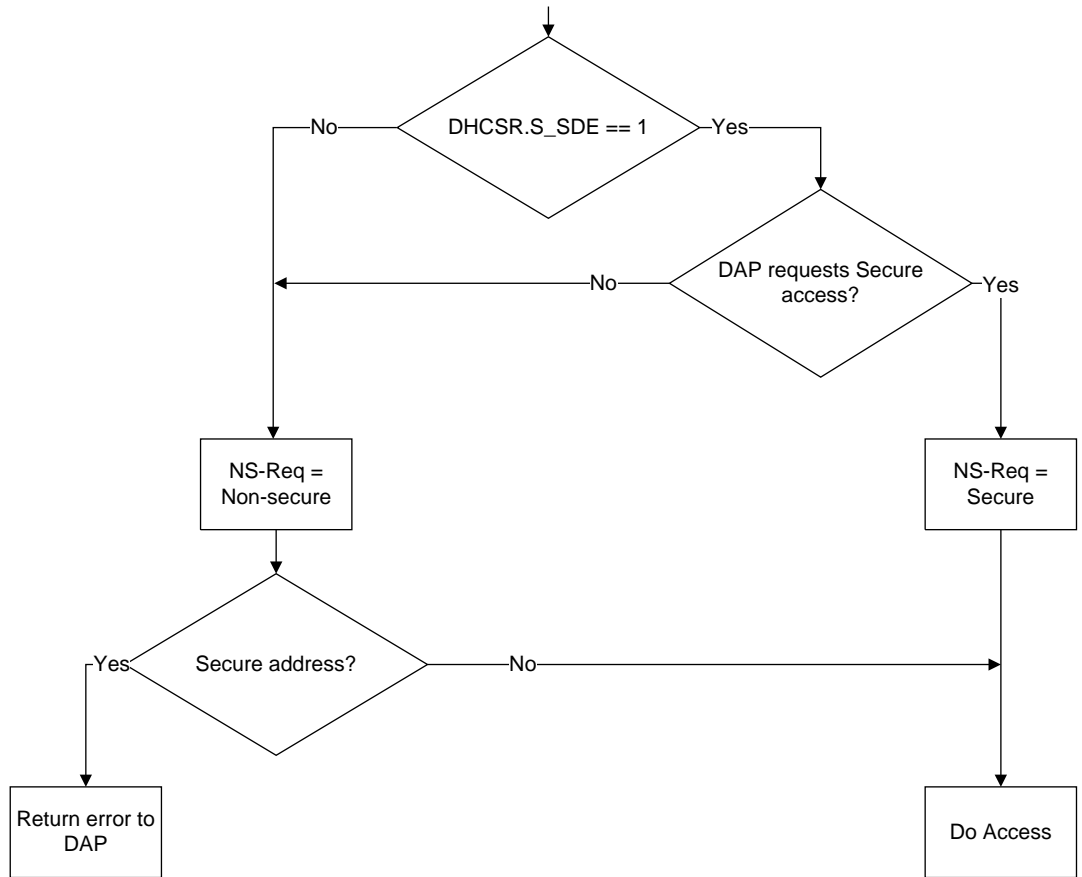
**R<sub>JHBC</sub>** If [DHCSR.S\\_SDE](#) == 1, and the DAP requests a Secure access, NS-Req is set to Secure.

The extension requirements are - [DB](#) && [S](#).

**R<sub>LVBG</sub>** If either [DHCSR.S\\_SDE](#) == 0 or the DAP requests a Non-secure access, NS-Req set to Non-secure.

The extension requirements are - [DB](#) && [S](#).

**R<sub>WMRR</sub>** DAP accesses are checked by the IDAU and the SAU, if applicable. That is, if NS-Req on a DAP access specifies Non-secure access, and the IDAU or SAU prohibits Non-secure access to the address, an error response is returned to the DAP.



The extension requirements are - **DB** && **S**.

**R<sub>VTTN</sub>** Unless otherwise stated DAP accesses are not checked by the MPU.

The extension requirements are - **DB** && **MPU**.

**R<sub>FDCQ</sub>** DAP accesses to the SCS registers ignore NS-Req.

The extension requirements are - **DB** && **S**.

**R<sub>SSVN</sub>** Permitted DAP accesses to Secure SCS registers in the range 0xE000E000-0xE000EFFF are affected by the values of **DHCSR.S\_SDE**, **DSCSR.SBRSELEN**, and **DSCSR.SBRSEL**, as well as by the current Security state of the PE. The following table shows the effect of these factors on the register being viewed.

| <b>DHCSR.S_SDE</b> | <b>DSCSR.SBRSELEN</b> | <b>DSCSR.SBRSEL</b> | <b>Current Security state of the PE</b> | <b>View of register accessed</b> |
|--------------------|-----------------------|---------------------|-----------------------------------------|----------------------------------|
| 0                  | X                     | X                   | X                                       | Non-secure.                      |
| 1                  | 1                     | 0                   | X                                       | Non-secure.                      |
| 1                  | 1                     | 1                   | X                                       | Secure.                          |
| 1                  | 0                     | X                   | Non-secure.                             | Non-secure.                      |
| 1                  | 0                     | X                   | Secure.                                 | Secure.                          |

The extension requirements are - **DB** && **S**.

Chapter B11. Debug

B11.3. Debug authentication interface

**R<sub>HXM</sub>G** Permitted DAP accesses to the region 0xE002E000-0xE002EFFF are RAZ/WI if the access is privileged and return an error if the access is unprivileged.

*The extension requirements are - **DB**.*

See also:

[B3.14 Secure address protection on page 81.](#)

[Chapter B8 The Armv8-M Protected Memory System Architecture on page 214.](#)

## B11.4 Debug event behavior

### B11.4.1 About debug events

**I<sub>CBWT</sub>** An event that is triggered for debug reasons is known as *a debug event*.

The extension requirements are - **DB**.

**R<sub>PQKW</sub>** A debug event that is not ignored causes one of the following to occur:

- If Halting debug is implemented and enabled, entry to Debug state.
- A HardFault exception.
- Lockup.

The extension requirements are - **DB**. Note, entry to Debug state requires Halting Debug.

**R<sub>QLTQ</sub>** A debug event that is not ignored, can cause a DebugMonitor exception to occur.

The extension requirements are - **M**.

**R<sub>MNKP</sub>** The HardFault exceptions or Lockup that are caused by debug events are generated by:

- A **BKPT** instruction that is executed when the PE can neither halt nor generate a DebugMonitor exception.
- In some circumstances, the FPB.

The extension requirements are - **M**. Note, an FPB requires FPB.

**R<sub>WCPW</sub>** The debug events are as follows.

| Debug event  | Actions                                     |
|--------------|---------------------------------------------|
| Step         | Halt or DebugMonitor exception.             |
| Halt Request | Halt                                        |
| Breakpoint   | Halt, DebugMonitor exception, or Hardfault. |
| Watchpoint   | Halt or DebugMonitor exception.             |
| Vector catch | Halt only                                   |
| External     | Halt or DebugMonitor exception.             |

Note, a DebugMonitor exception requires M. Halt requires Halting Debug.

**R<sub>LDRZ</sub>** The **DFSR** contains status bits for each debug event. These bits are set to 1 when a debug event causes the PE to halt or generate a DebugMonitor exception, and are then write-one-to-clear.

The following table shows which bit is set for each debug event.

| Event cause  | DFSR bit        |
|--------------|-----------------|
| Step         | <b>HALTED</b>   |
| Halt request | <b>HALTED</b>   |
| Breakpoint   | <b>BKPT</b>     |
| Watchpoint   | <b>DWTTRAP</b>  |
| Vector catch | <b>VCATCH</b>   |
| External     | <b>EXTERNAL</b> |

The extension requirements are - **M** || Halting debug.

**R<sub>HNRV</sub>** It is IMPLEMENTATION DEFINED whether the **DFSR** debug event bits are updated when an event is ignored.

The extension requirements are - **DB**.

**I<sub>NSMV</sub>** Debug events are either synchronous or asynchronous.

The extension requirements are - **DB**.

- R<sub>VSVN</sub>** The synchronous debug events are:
- Breakpoint debug events, caused by execution of a **BKPT** instruction or by a match in the FPB.
  - Vector catch debug events, caused when one or more **DEMCR.VC\_\*** bits are set to 1, and the PE takes the corresponding exception.
  - Step debug events, caused by **DHCSR.C\_STEP** or **DEMCR.MON\_STEP**.
- The extension requirements are - **DB**.*
- R<sub>PVGM</sub>** A single instruction can generate several synchronous debug events.
- The extension requirements are - **DB**.*
- R<sub>WJFB</sub>** Synchronous debug events are associated with the instruction that generated them and are taken instead of executing the instruction. The PE does not generate any other synchronous exception or debug event that might have occurred as a result of executing the instruction.
- The extension requirements are - **DB**.*
- R<sub>RNRD</sub>** The Step debug event is taken on the instruction following the instruction being stepped. This means that prioritization of the event applies relative to any other exception or debug event for the following instruction, not for the instruction being stepped.
- The extension requirements are - **DB**.*
- R<sub>JSPS</sub>** If multiple synchronous debug events and exceptions are generated on the same instruction, they are prioritized as follows:
1. Halt request (halting only), including where **DHCSR.C\_HALT** is set by **DHCSR.C\_STEP** of the previous instruction.
  2. Highest-priority pending exception that is eligible to be taken. If the Main Extension is implemented, this might be a DebugMonitor exception, if **DEMCR.MON\_PEND** == 1. This includes where **DEMCR.MON\_PEND** is set by **DEMCR.MON\_STEP** of the previous instruction.
  3. Vector catch.
  4. Fault from an instruction fetch, including synchronous BusFault error.
  5. Breakpoint that is signaled by an FPB unit.
  6. **BKPT** instruction or other exception that results from decoding the instructions. This includes the cases where exceptions from the instruction are UNDEFINED, an unimplemented or disabled coprocessor is targeted, or the **EPSR.T** bit has a value of 1.
  7. Other synchronous exception that is generated by executing the instruction, including an exception that is generated by a memory access that is generated by the instruction.
- The extension requirements are - **DB**. Note, not all of the debug features listed might be implemented in a particular implementation.*
- R<sub>BQVF</sub>** The highest-priority synchronous debug event is reported in the **DFSR**.
- The extension requirements are - **DB**.*
- R<sub>FWQQ</sub>** It is UNPREDICTABLE whether synchronous debug events that occur on the same instruction as a debug event with a higher priority are reported in the **DFSR**.
- The extension requirements are - **DB**.*
- R<sub>TKRS</sub>** The asynchronous debug events are:
- Watchpoint debug events caused by a match in the DWT, including instruction address match watchpoints.
  - Halt request debug events, where either:
    - A debugger write that has set **DHCSR.C\_HALT** to 1 and **DHCSR.C\_DEBUGEN** set to 1.
    - A software write that sets **DHCSR.C\_HALT** to 1 when **DHCSR.C\_DEBUGEN** was set to 1.

- External debug request debug events caused by assertion of an IMPLEMENTATION DEFINED external debug request.

The extension requirements are - *DB*.

**R<sub>MRC</sub>** When `DHCSR.C_DEBUGEN == 0` or the PE is in a state in which halting is prohibited, `DHCSR.C_HALT` and `DHCSR.C_STEP` are ignored, and these bits have an *Effective value* of 0.

The extension requirements are - *Halting debug*.

See also:

[B3.13 Priority model on page 77](#).

[Halting debug](#).

[DebugMonitor exception](#).

[B11.4.3 Vector catch on page 259](#).

[GenerateDebugEventResponse\(\)](#).

## Halting debug

**R<sub>WLCE</sub>** Setting the `DHCSR.C_DEBUGEN` bit to 1 enables Halting debug.

The extension requirements are - *Halting debug*.

**R<sub>RZTG</sub>** A debug event sets `DHCSR.C_HALT` to 1 if all of the following conditions apply:

- The debug event supports generating entry to Debug state.
- `DHCSR.C_DEBUGEN == 1`.
- Unless otherwise stated, halting is allowed.

The extension requirements are - *Halting debug*.

**R<sub>THLS</sub>** If `DHCSR.C_HALT` has a value of 1 and halting is allowed, the PE halts and enters Debug state.

The extension requirements are - *Halting debug*.

**R<sub>FKWB</sub>** A debug event that sets `DHCSR.C_HALT` to 1 pends entry to Debug state.

The extension requirements are - *Halting debug*.

**I<sub>VBXN</sub>** A debug event might set `DHCSR.C_HALT` and remain pending through execution in a mode or state where Halting debug is prohibited, which might not be a finite time.

The extension requirements are - *Halting debug*.

**R<sub>HQTX</sub>** When `SecureHaltingDebugEnabled()` returns FALSE and `HaltingDebugEnabled()` returns TRUE, a Debug event that sets `DHCSR.C_HALT` to 1 will cause all of the following:

- The PE to halt on transition from Secure state to Non-secure state for any of the following reasons:
  - Exception entry.
  - Exception return.
  - Non-secure function call.
  - Function return.
- The PE will halt and enter Debug state before completing the first instruction in Non-secure state.

The extension requirements are - *Halting debug*.



- R<sub>XSRJ</sub>** If **DHCSR.C\_HALT** has a value of 1 or **EDBGRQ** is asserted before a **Context synchronization event**, and halting is allowed after the **Context synchronization event**, then the PE halts and enters Debug state before the first instruction following the Context synchronization event completes its execution.  
*The extension requirements are - Halting debug || EDBGRQ.*
- R<sub>JXQF</sub>** **DFSR** is updated at the same time as the PE sets **DHCSR.C\_HALT** to 1.  
*The extension requirements are - Halting debug.*
- R<sub>TXWB</sub>** If an instruction that is being stepped or an instruction that generates a debug event reads **DFSR** or **DHCSR**, the value that is read for the relevant **DFSR** bit or for **DHCSR.C\_HALT** is UNKNOWN.  
*The extension requirements are - Halting debug.*
- R<sub>FRJC</sub>** For asynchronous debug events, if halting is allowed, the PE enters Debug state in finite time.  
*The extension requirements are - Halting debug.*
- R<sub>VJKX</sub>** Entering Debug state has no architecturally defined effect on the Event Register and exclusive monitors.  
*The extension requirements are - Halting debug.*
- I<sub>JNGH</sub>** **DHCSR.C\_SNAPSTALL** might allow imprecise entry into the Debug state, for example by forcing any stalled load or store instructions to be abandoned.  
*The extension requirements are - Halting debug.*
- R<sub>BTBJ</sub>** If **DHCSR.C\_DEBUGEN** == 0 or **HaltingDebugAllowed()** == FALSE, **DHCSR.C\_SNAPSTALL** is ignored and has an **Effective value** of 0.  
*The extension requirements are - Halting debug.*
- R<sub>HLNF</sub>** If **DHCSR.S\_SDE** == 0, **DHCSR.C\_SNAPSTALL** ignores writes from the debugger.  
*The extension requirements are - Halting debug && S.*
- R<sub>RKBJ</sub>** When the PE is in a state in which halting is prohibited, if **DHCSR.C\_HALT** == 1 and **DHCSR.C\_DEBUGEN** == 1, then **DHCSR.C\_HALT** remains set unless it is cleared by a direct write to **DHCSR**. If the PE enters a state in which halting is allowed while **DHCSR.C\_HALT** is set to 1, then the PE enters Debug state.  
*The extension requirements are - Halting debug.*

See also:

[DHCSR, Debug Halting Control and Status Register.](#)

[B11.4.2 Debug stepping on page 255.](#)

[B11.5 Debug state on page 263.](#)

## DebugMonitor exception

- I<sub>DPCC</sub>** The DebugMonitor exception is only available if the Main Extension is implemented.  
*The extension requirements are - M.*
- R<sub>ZBSJ</sub>** The DebugMonitor exception is enabled when the **DEMCR.MON\_EN** bit is set to 1.  
*The extension requirements are - M.*
- R<sub>PPLF</sub>** A debug event sets **DEMCR.MON\_PEND** to 1 if all of the following conditions apply:

- The debug event supports generating DebugMonitor exceptions and does not generate an entry to Debug state.
- `DEMCR.MON_EN == 1`.
- `DEMCR.SDME == 1` for Secure state DebugMonitor exceptions.
- The DebugMonitor exception group priority is greater than the current execution priority.

The function `CanPendMonitorOnEvent()` describes this.

The extension requirements are - *M*.

**R<sub>XNMW</sub>** If a Debug event does not generate an entry to Debug state and `DEMCR.MON_EN` is set to 0, or the DebugMonitor exception group priority value is lower than the current execution priority, or `DEMCR.SDME == 0` and the DebugMonitor exception was generated in Secure state:

- The PE escalates a DebugMonitor synchronous exception that is generated by executing a `BKPT` instruction to a `HardFault`.
- The PE might set `DEMCR.MON_PEND` to 1 for a watchpoint debug event.
- The PE ignores the other debug events.

The extension requirements are - *M*.

**R<sub>CHXQ</sub>** A debug event that sets `DEMCR.MON_PEND` to 1 pends a DebugMonitor exception.

The extension requirements are - *M*.

**R<sub>VSPX</sub>** `DEMCR.MON_PEND` is cleared to 0 when the PE takes a DebugMonitor exception. This means that a value of 1 for `DEMCR.MON_PEND` might never be observed for a synchronous DebugMonitor exception.

The extension requirements are - *M*.

**R<sub>BRXT</sub>** `DFSR` is updated at the same time as the PE sets `DEMCR.MON_PEND` to 1.

The extension requirements are - *M*.

**R<sub>BKHP</sub>** If an instruction that is being stepped or that generates a debug event reads `DFSR` or `DEMCR`, the value that is read for the relevant `DFSR` bit or for `DEMCR.MON_PEND` is UNKNOWN.

The extension requirements are - *M*.

**R<sub>VFLQ</sub>** For asynchronous debug events, if taken as a DebugMonitor exception, and if the current priority is lower than the DebugMonitor exception group priority, a DebugMonitor exception is taken in finite time.

The extension requirements are - *M*.

**R<sub>JVSC</sub>** A direct write to `DEMCR` can set `DEMCR.MON_PEND` to 1 at any time to make the DebugMonitor exception pending or can set `DEMCR.MON_PEND` to 0 to remove a pending DebugMonitor exception.

The extension requirements are - *M*.

**R<sub>XPBN</sub>** When `DEMCR.MON_PEND == 1`, the PE takes the DebugMonitor exception according to the exception prioritization rules, regardless of the value of `DEMCR.SDME` and `DEMCR.MON_EN`.

The extension requirements are - *M*.

**R<sub>QZHW</sub>** Unless otherwise stated, synchronous and asynchronous DebugMonitor exceptions can only cause preemption at instruction boundaries.

The extension requirements are - *M*.

**I<sub>PJJD</sub>** DebugMonitor exceptions cannot cause instruction resume or instruction restart. However, if another exception preempts an execution-continuable instruction that also generates a watchpoint, the PE might take that exception

during the instruction, or abandon the instruction to take the exception, and, after returning from the exception, tail-chain to the DebugMonitor exception.

The extension requirements are - **M**.

See also:

[B12.2.2 DWT unit operation on page 279.](#)

[B12.5.2 FPB unit operation on page 301.](#)

[B3.27 Exceptions, instruction resume, or instruction restart on page 111.](#)

## B11.4.2 Debug stepping

**R<sub>HMCN</sub>** The Armv8-M architecture supports debug stepping in both Halting debug and for the DebugMonitor exception.

The extension requirements are - **Halting debug** || **M**. Note, might require the DebugMonitor exception.

**R<sub>THTG</sub>** It is IMPLEMENTATION DEFINED whether stepping a **WFE** or **WFI** instruction causes the **WFE** or **WFI** instruction to:

- Retire and take the debug event.
- Go into a sleep state and take the debug event only when another wake up event occurs.

The extension requirements are - **Halting debug** || **M**.

**R<sub>LLVC</sub>** If a debug event wakes a **WFE** or **WFI** instruction, then on taking the debug event, the instruction has retired.

The extension requirements are - **Halting debug** || **M**.

See also:

[Halting debug stepping.](#)

[Debug monitor stepping.](#)

### Halting debug stepping

**I<sub>QMXC</sub>** A debugger can use Halting debug stepping to exit from Debug state, execute a single instruction, and then reenter Debug state.

The extension requirements are - **Halting debug**.

**R<sub>SWKC</sub>** Halting debug stepping is active when all of the following apply:

- **DHCSR.C\_DEBUGEN** is set to 1, Halting debug is enabled, and halting is allowed.
- **DHCSR.C\_STEP** is set to 1, halting stepping is enabled.
- The PE is in Non-debug state.

The extension requirements are - **Halting debug**.

**R<sub>ZVKS</sub>** When the PE exits Debug state and Halting debug stepping becomes active, the PE performs a Halting debug step as follows:

1. Performs one of the following:

- Completes the next instruction without generating any exception.
- Takes any pending exception entry of sufficient priority, without completing the next instruction. The PE performs an exception entry sequence that stacks the next instruction context. This context might include instruction continuation bits if the next instruction was partly executed and supports instruction resume. The exception might be a pending exception, or an exception generated by the execution of the

next instruction.

- Completes the execution of the next instruction, and then takes any pending exception of sufficient priority. The PE performs an exception entry sequence that stacks the following instruction context.
- If the next instruction is an exception return instruction, completes the next instruction, tail-chaining to enter a new exception handler.

In each case where the PE performs an exception entry sequence it does so according to the exception priority and late-arrival rules, meaning derived and late-arriving exceptions might preempt the exception entry sequence.

The exception behavior is not recursive. Only a single `PushStack()` update can occur in a step sequence.

2. Sets `DFSR.HALTED` and `DHCSR.C_HALT` to 1. A read of the `DFSR.HALTED` or the `DHCSR.C_HALT` bit performed by the stepped instruction returns an UNKNOWN value.
3. After the Halting debug step, before executing the following instruction, because `DHCSR.C_HALT` is set the PE will halt and enter Debug state if halting is still allowed. However, if halting is prohibited after the Halting debug step then the PE does not enter Debug state and `DHCSR.C_HALT` remains set.

*The extension requirements are - Halting debug.*

**I<sub>LTRX</sub>** The debugger can optionally set **DHCSR.C\_MASKINTS** to 1 to prevent PENDSV, SysTick, and external configurable interrupts from being taken. If a permitted exception becomes active, the PE steps into the exception handler and halts before executing the first instruction of the associated exception handler.

*The extension requirements are - Halting debug.*

**R<sub>ZDYR</sub>** If **DHCSR.C\_DEBUGEN** == 0 or **HaltingDebugAllowed()** == FALSE, **DHCSR.C\_MASKINTS** is ignored and has an **Effective value** 0.

*The extension requirements are - Halting debug.*

**R<sub>FWSN</sub>** If **DHCSR.S\_SDE** == 0, **DHCSR.C\_MASKINTS** is ignored for exceptions targeting Secure state.

*The extension requirements are - Halting debug && S.*

**R<sub>MBCB</sub>** **DHCSR**.{**C\_HALT**, **C\_STEP**, **C\_MASKINTS**} can be written in a single write to **DHCSR**, as follows:

| <b>DHCSR write</b>                                                                                                        |               |                   |                                                                                                                                                                                                              |
|---------------------------------------------------------------------------------------------------------------------------|---------------|-------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| assumes that <b>DHCSR.C_DEBUGEN</b> and <b>DHCSR.S_HALT</b> are both set to 1 when the write occurs and the PE is halted. |               |                   |                                                                                                                                                                                                              |
| <b>C_HALT</b>                                                                                                             | <b>C_STEP</b> | <b>C_MASKINTS</b> | <b>Effect</b>                                                                                                                                                                                                |
| 0                                                                                                                         | 0             | 0                 | Exit Debug state and start instruction execution. Exceptions can become active and prioritized according to the priority rules and the configuration of exceptions.                                          |
| 0                                                                                                                         | 0             | 1                 | Exit Debug state and start instruction execution. PendSV, SysTick and, external configurable interrupts are disabled, otherwise exceptions can become active and proritized according to the priority rules. |
| 0                                                                                                                         | 1             | 0                 | Exit Debug state, step an instruction and halt. Exceptions can become active and prioritized according to the priority rules.                                                                                |
| 0                                                                                                                         | 1             | 1                 | Exit Debug state, step an instruction and halt. PendSV, SysTick and, external configurable interrupts are disabled, otherwise exceptions can become active and proritized according to the priority rules.   |
| 1                                                                                                                         | X             | X                 | Remain in Debug state.                                                                                                                                                                                       |

The write to **DHCSR** assumes that **DHCSR.C\_DEBUGEN** and **DHCSR.S\_HALT** are both set to 1 when the write occurs, meaning the PE is halted.

*The extension requirements are - Halting debug.*

## Debug monitor stepping

**I<sub>DXCT</sub>** A debugger can use debug monitor stepping to return from the DebugMonitor exception handler, execute a single instruction, and then reenter the DebugMonitor exception handler.

*The extension requirements are - DebugMonitor exception.*

**R<sub>MLRM</sub>** Debug monitor stepping is active when all of the following apply:

- **DHCSR.C\_DEBUGEN** is set to 0 or the PE is in a state in which halting is prohibited.
- **DEMCR.MON\_EN** is set to 1, that is Monitor debug is enabled.
- **DEMCR.MON\_STEP** is set to 1, that is monitor stepping is enabled.
- **DEMCR.SDME** == 1 or the PE is in Non-secure state.
- Execution priority is below the priority of the DebugMonitor exception.

*The extension requirements are - M.*

**R<sub>MWFT</sub>** When DebugMonitor stepping becomes active, the PE performs a DebugMonitor step as follows:

1. It performs one of the following:

- It completes the next instruction without generating any exception.
- It takes any pending exception of sufficient priority. The PE performs an exception entry sequence that stacks the next instruction context. The exception might be a pending exception, or it might be an exception generated by the execution of the next instruction.
- If the next instruction is an exception return instruction, the PE completes the next instruction, tail-chaining to enter a new exception handler according to the normal exception priority and late-arrival rules.

If the PE performs an exception entry sequence as part of step 1, the PE stacks the next instruction context. This context might include instruction continuation bits if the next instruction was partly executed and

supports instruction resume.

2. If the execution priority is below the priority of the DebugMonitor exception after step 1, the PE sets `DEMCR.MON_PEND` and `DFSR.HALTED` to 1. A read of `DEMCR.MON_PEND` or `DFSR.HALTED` by the stepped instruction returns an UNKNOWN value.
3. Before executing the following instruction, the PE takes any pending exception with sufficient priority.

If step 2 set `DEMCR.MON_PEND` to 1, then the DebugMonitor exception is pending. However, it is UNPREDICTABLE whether the PE uses the new value or the old value of `DEMCR.MON_PEND` in determining the highest priority exception. This means that:

- Another exception might preempt execution before the DebugMonitor exception is taken, and the exception might be lower priority than the DebugMonitor exception. However, this is a [Context synchronization event](#) and the PE uses the new value of `DEMCR.MON_PEND` to determine the highest priority exception before executing the next instruction.
- If no other exceptions are pending, the PE takes the DebugMonitor exception.

Derived and late-arriving exceptions might preempt the exception entry sequence.

*The extension requirements are - M.*

**I<sub>GPSX</sub>** In all other cases, the DebugMonitor exception preempting execution returns control to the DebugMonitor exception handler. Unless that handler clears `DEMCR.MON_STEP` to 0, returning from the handler performs the next debug monitor step.

*The extension requirements are - M.*

**I<sub>KPKX</sub>** If, after the debug monitor stepping process, the taking of an exception means that the execution priority is no longer below that of the DebugMonitor exception, the values of `DEMCR.MON_STEP` and `DEMCR.MON_PEND` mean that debug monitor stepping process continues when execution priority falls back below the priority of the DebugMonitor exception.

*The extension requirements are - M.*

### B11.4.3 Vector catch

**I<sub>TVRX</sub>** Vector catch is the mechanism for generating a debug event and entering Debug state on entry to a particular exception handler or reset.

*The extension requirements are - M.*

**R<sub>JCXR</sub>** Vector catching is only supported by Halting debug.

*The extension requirements are - Halting debug.*

**R<sub>PBVX</sub>** The conditions for a vector catch, other than reset vector catch, are:

- `DHCSR.C_DEBUGEN` == 1 and halting is allowed for the Security state the exception is targeting.
- The associated exception enable bit is set.
- The associated active bit is set.
- The associated vector catch enable bit.
- An exception is taken to the relevant exception handler. The associated fault status register status bit is set to 1.

When these conditions are met, the PE sets `DHCSR.C_HALT` to 1 and enters Debug state before executing the first instruction of the exception handler.

*The extension requirements are - Halting debug. Note, If the Main Extension is not implemented only bits [24],[10] and [0] of DEMCR are implemented with Halting debug functionality. SecureFault requires S.*

Chapter B11. Debug  
B11.4. Debug event behavior

**I<sub>XDGP</sub>** Late arrival and derived exceptions might occur, preempting the exception targeted by the vector catch and postponing when the PE halts.

The extension requirements are - *Halting debug*.

**R<sub>XKMH</sub>** The following table defines the exception, Fault status bit, and Vector catch bit.

| Exception       | Fault status bit | Vector catch bit<br>DEMCR |
|-----------------|------------------|---------------------------|
| HardFault       | HFSR.VECTBL      | VC_INTERR                 |
|                 | HFSR.FORCED      | VC_HARDERR                |
|                 | HFSR.DEBUGEVT    | VC_HARDERR                |
| BusFault        | BFSR.IBUSERR     | VC_BUSERR                 |
|                 | BFSR.PRECISERR   | VC_BUSERR                 |
|                 | BFSR.IMPRESISERR | VC_BUSERR                 |
|                 | BFSR.UNSTKERR    | VC_INTERR                 |
|                 | BFSR.STKERR      | VC_INTERR                 |
|                 | BFSR.LSPERR      | VC_INTERR                 |
| DebugMonitor    | HFSR.DEBUGEVT    | -                         |
| MemManage fault | MMFSR.IACCVIOL   | VC_MMERR                  |
|                 | MMFSR.DACCVIOL   | VC_MMERR                  |
|                 | MMFSR.MUNSTKERR  | VC_INTERR                 |
|                 | MMFSR.MSTKERR    | VC_INTERR                 |
|                 | MMFSR.MLSPERR    | VC_INTERR                 |
| NMI             | -                | -                         |
| PENDSV          | -                | -                         |
| UsageFault      | UFSR.UNDEFINSTR  | VC_STATERR                |
|                 | UFSR.INVSTATE    | VC_STATERR                |
|                 | UFSR.INVPC       | VC_STATERR                |
|                 | UFSR.NOCP        | VC_NOCPERR                |
|                 | UFSR.STKOF       | VC_INTERR                 |
|                 | UFSR.UNALIGNED   | VC_CHKERR                 |
|                 | UFSR.DIVBYZERO   | VC_CHKERR                 |
|                 | SecureFault      | SFSR.INVEP                |
| SFSR.INVIS      |                  | VC_SFERR                  |
| SFSR.INVER      |                  | VC_SFERR                  |
| SFSR.AUVIOL     |                  | VC_SFERR                  |
| SFSR.INVTRAN    |                  | VC_SFERR                  |
| SFSR.LSPERR     |                  | VC_SFERR                  |
| SFSR.LSERR      |                  | VC_SFERR                  |
| SVCall          | -                | -                         |
| SysTick         | -                | -                         |

The extension requirements are - *M*.

**R<sub>LKNL</sub>** When **DHCSR.C\_DEBUGEN** == 0 or the PE is in a state in which halting is prohibited, all **DEMCR.VC\_** bits, other than **DEMCR.VC\_CORERESSET**, are ignored.

The extension requirements are - *Halting debug && S*.

**R<sub>WRMQ</sub>** The PE pends a Vector catch event when all of the following is true:

- The PE has reset into Secure state.
- **DHCSR.C\_DEBUGEN** == 1.
- **DEMCR.VC\_CORERESSET** == 1.
- Halting debug is not allowed in Secure state.



The PE does not halt until either it enters Non-secure state or debug is allowed in Secure state.

The extension requirements are - *Halting debug* && *S*.

See also:

[B1.1 Resets, Cold reset, and Warm reset](#) on page 47.

[B3.10 Exception enable, pending, and active bits](#) on page 68.

[B3.13 Priority model](#) on page 77.

[B3.12 Faults](#) on page 72.

[B3.9 Exception numbers and exception priority numbers](#) on page 65.

[B3.24 Exceptions during exception entry](#) on page 105.

[B3.25 Exceptions during exception return](#) on page 107.

[Chapter B1 Resets](#) on page 46.

## B11.4.4 Breakpoint instructions

**R<sub>CRJG</sub>** When `DHCSR.C_DEBUGEN == 0` or when the PE is in a state in which halting is prohibited, the `BKPT` instruction does not generate an entry to Debug state. If no DebugMonitor exception is generated, the `BKPT` instruction generates a HardFault exception or enters Lockup state.

**R<sub>MFHN</sub>** A `BKPT` instruction halts the PE if all of the following conditions apply:

- `HaltingDebugAllowed() == TRUE`.
- `DHCSR.C_DEBUGEN == 1`.
- The Security Extension is not implemented, the instruction is executed in Non-secure state, or `DHCSR.S_SDE == 1`.

The extension requirements are - *Halting debug*.

**R<sub>FLKK</sub>** A `BKPT` instruction generates a DebugMonitor exception if it does not halt the PE and all of the following conditions apply:

- `DEMCR.MON_EN == 1`.
- The DebugMonitor exception group priority is greater than the current execution priority.
- The Security Extension is not implemented, the instruction is executed in Non-secure state, or `DEMCR.SDME == 1`.

The extension requirements are - *M*.

## B11.4.5 External debug request

**R<sub>XZCP</sub>** When the PE is in Non-debug state, an external agent can signal an external debug request.

**R<sub>GTGX</sub>** An external debug request can cause a debug event, that causes either:

- Entry to Debug state.
- If the Main Extension is implemented, a DebugMonitor exception.

The extension requirements are - *M* || *Halting debug*.

Chapter B11. Debug

B11.4. Debug event behavior

- R<sub>FGCV</sub>** The PE ignores external debug requests when it is in Debug state.  
*The extension requirements are - **Halting debug**.*
- R<sub>BXRD</sub>** When **DHCSR.C\_DEBUGEN** == 0 or the PE is in a state in which halting is prohibited, an External debug request does not generate an entry to Debug state and is ignored if no DebugMonitor exception is generated.
- R<sub>WGMB</sub>** If the DebugMonitor exception group priority is greater than the current execution priority and **DEMCR.MON\_EN** == 1, an External debug request that does not generate an entry to Debug state sets **DEMCR.MON\_PEND** to 1.  
*The extension requirements are - **M**.*

See also:

[B11.4 Debug event behavior on page 250.](#)

[DFSR.EXTERNAL.](#)

## B11.5 Debug state

**R<sub>RMKS</sub>** In Halting debug, debug events allow an external debugger to halt the PE. The PE then enters Debug state. When the PE is in Debug state:

- The PE stops executing instructions from the location indicated by the PC, and is instead controlled by the external debug interface.
- The PE cannot service any interrupts.

*The extension requirements are - Halting debug.*

**R<sub>QDCP</sub>** In Debug state, the PE clears the **DHCSR.S\_REGRDY** bit to 0 when the debugger writes to **DCRSR** and the PE then sets the bit to 1 when the transfer between the **DCRDR** and R0-R12 (**Rn**), Special-purpose register, Floating-point Extension register, or DebugReturnAddress completes.

*The extension requirements are - Halting debug. Note, Floating-point registers are RES0 if FP is not implemented.*

**I<sub>FKSM</sub>** To transfer a word to a general-purpose register, to a Special-purpose register, to a Floating-point Extension register, or to DebugReturnAddress, a debugger:

1. Writes the required word to **DCRDR**.
2. Writes to the **DCRSR**, with the REGSEL value indicating the required register, and the REGWnR bit set to 1 to indicate a write access. This clears the **DHCSR.S\_REGRDY** bit to 0.
3. If required, polls **DHCSR** until **DHCSR.S\_REGRDY** reads-as-one. This shows that the PE has transferred the **DCRDR** value to the selected register.

*The extension requirements are - Halting debug.*

**I<sub>CMBB</sub>** To transfer a word from a general-purpose register, from a Special-purpose register, from a Floating-point Extension register, or from DebugReturnAddress, a debugger:

1. Writes to **DCRSR**, with the REGSEL value indicating the required register, and the REGWnR bit as 0 to indicate a read access. This clears the **DHCSR.S\_REGRDY** bit to 0.
2. Polls **DHCSR** until **DHCSR.S\_REGRDY** reads-as-one. This shows that the PE has transferred the value of the selected register to **DCRDR**.
3. Reads the required value from **DCRDR**.

*The extension requirements are - Halting debug.*

**R<sub>VLVD</sub>** In Debug state, following a write to **DCRSR** that clears the **DHCSR.S\_REGRDY** bit to 0, the behavior is UNPREDICTABLE if any of the following occur before the PE sets **DHCSR.S\_REGRDY** to 1:

- The PE exits Debug state, other than because of a Warm reset.
- The debugger writes to **DCRDR** or **DCRSR**.

If the **DCRSR.REGWnR** bit was set to 0 and the debugger reads from **DCRDR** before the PE sets **DHCSR.S\_REGRDY** to 1, then the read returns an UNKNOWN value.

*The extension requirements are - Halting debug.*

**R<sub>JKBB</sub>** When using the **DCRDR**, **DCRSR** and **DHCSR.S\_REGRDY** mechanism to write to **XPSR**, all bits of the **XPSR** are fully accessible. The effect of writing an illegal value is UNPREDICTABLE.

*The extension requirements are - Halting debug.*

**I<sub>RXQB</sub>** The **DCRDR**, **DCRSR** and **DHCSR.S\_REGRDY** mechanism differs from the behavior of MSR or MRS instruction accesses to the **XPSR**, where some bits are ignored on writes.

*The extension requirements are - Halting debug.*

- R<sub>QLRN</sub>** When the PE is halted the Debugger can write to:
- The DebugReturnAddress.
  - **EPSR.IT/ICI** bits.
- On exiting Debug state the PE starts from DebugReturnAddress. The Debugger must ensure that the **EPSR.IT** and **EPSR.ICI** bits are consistent with DebugReturnAddress, otherwise instruction execution will be UNPREDICTABLE.
- The extension requirements are - Halting debug.*
- I<sub>RRFN</sub>** The debugger can always set **FAULTMASK** to 1, but doing so might cause unexpected behavior on exit from Debug state. An MSR instruction cannot set **FAULTMASK** to 1 when the execution priority is -1 or higher.
- The extension requirements are - Halting debug.*
- R<sub>XRRQ</sub>** The debugger can write to the **EPSR.IT/ICI** bits, and on exiting Debug state any interrupted LDM or STM instruction will use these new values. Clearing the ICI bits to 0 will cause the interrupted LDM or STM instruction to restart or continue.
- The extension requirements are - Halting debug.*
- R<sub>BMHD</sub>** When the PE is in Debug state, an indirect write to a Special-purpose register caused by an access by the debugger to a register within the System Control Block (SCB) is guaranteed to be visible after the access to the register within the SCB completed to a subsequent:
- Access to the Special-purpose register through **DCRDR**.
  - Indirect read of the Special-purpose register made for an access of any register through **DCRDR** or any register within the System Control Block.
- The extension requirements are - Halting debug.*
- R<sub>MDJX</sub>** When the PE is in Debug state, a write to a Special-purpose register made by the debugger through the **DCRDR** is guaranteed to be visible after the write is observed to be completed in **DHCSR.S\_REGRDY** to a subsequent:
- Access of any register through **DCRDR** or any register within the System Control Block.
  - Indirect read of the Special-purpose register made for an access to any register through **DCRDR** or any register within the System Control Block.
- The extension requirements are - Halting debug.*
- I<sub>DMTG</sub>** A read or write of a register through **DCRDR** starts with a write to **DCRSR**. Where the architecture guarantees that a previous access is visible to a subsequent access through **DCRDR**, this means the write to **DCRSR** is made after the point where the previous access is visible.
- The extension requirements are - Halting debug.*

See also:

*DCRDR, Debug Core Register Data Register.*

*DCRSR, Debug Core Data Select Register.*

## B11.6 Exiting Debug state

- R<sub>BFGT</sub>** The PE exits Debug state:
- When the debugger writes 0 to [DHCSR.C\\_HALT](#).
  - On receipt of an external restart request.
  - On Warm reset.

*The extension requirements are - Halting debug.*

- R<sub>GGMJ</sub>** For synchronous debug events DebugReturnAddress is:

| Synchronous debug event                                      | DebugReturnAddress                                                                                                                                                                                                                 |
|--------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Breakpoint debug events ( <a href="#">BKPT</a> or FPB Match) | Address of the breakpointed instruction.                                                                                                                                                                                           |
| Vector Catch debug events                                    | Address of the first instruction of the exception handler.                                                                                                                                                                         |
| Step debug events                                            | Address of the next instruction to be executed in simple sequential execution order following the instruction that was stepped. If an exception was taken during stepping, this is the first instruction of the exception handler. |

*The extension requirements are - Halting debug.*

- R<sub>XCCB</sub>** Bit[0] of a DebugReturnAddress value is RAZ/SBZ. When writing a DebugReturnAddress, writing bit [0] of the address does not affect the [EPSR.T](#) bit.

*The extension requirements are - Halting debug.*

- R<sub>HNKB</sub>** Exiting Debug state has no architecturally defined effect on the Event Register and exclusive monitors.

*The extension requirements are - Halting debug.*

- R<sub>WKSD</sub>** If software clears [DHCSR.C\\_HALT](#) to 0 when the PE is in Debug state, a subsequent read of the [DHCSR](#) that returns 1 for both [DHCSR.C\\_HALT](#) and [DHCSR.S\\_HALT](#) indicates that the PE has reentered Debug state because it has detected a new debug event.

*The extension requirements are - Halting debug.*

- R<sub>FKXH</sub>** Before leaving Debug state caused by an imprecise entry into Debug state the system is reset.

*The extension requirements are - Halting debug.*

See also:

[B11.5 Debug state on page 263](#)

## B11.7 Multiprocessor support

**R<sub>QXLS</sub>** Systems that support debug of more than one PE, either within a single device or as heterogeneous PEs in a more complex system, require each PE to support all of the following to enable cross-triggering of debug events between PEs:

- An external debug request.
- A cross-halt event.
- An external restart request.

Support for these features is OPTIONAL in other systems.

### B11.7.1 Cross-halt event

**R<sub>DLCV</sub>** When the PE enters Debug state, it signals to an external agent that it is entering Debug state.  
*The extension requirements are - **Halting debug**.*

### B11.7.2 External restart request

**R<sub>ZKVV</sub>** When the PE is in Debug state, an external agent can signal an external restart request that causes the PE to exit Debug state.  
*The extension requirements are - **Halting debug**.*

**R<sub>WJST</sub>** An external restart request is not ordered with respect to accesses to memory-mapped registers. It is UNPREDICTABLE whether an access to a memory-mapped register from a DAP completes before an external restart request.  
*The extension requirements are - **Halting debug**.*

**I<sub>VNDK</sub>** A debugger ensures that any read or write of a memory-mapped register by the DAP completes before issuing an external restart request.  
*The extension requirements are - **DB**.*

**R<sub>NJQN</sub>** The PE ignores external restart requests when it is in Non-debug state.

See also:

[B11.6 Exiting Debug state on page 266.](#)

## Chapter B12

# Debug and Trace Components

This chapter specifies the Armv8-M debug and trace component rules. It contains the following sections:

[B12.1 Instrumentation Trace Macrocell on page 269.](#)

[B12.2 Data Watchpoint and Trace unit on page 278.](#)

[B12.3 Embedded Trace Macrocell on page 298.](#)

[B12.4 Trace Port Interface Unit on page 299.](#)

[B12.5 Flash Patch and Breakpoint unit on page 301.](#)



## B12.1 Instrumentation Trace Macrocell

### B12.1.1 About the ITM

**R<sub>GDNG</sub>** The *Instrumentation Trace Macrocell (ITM)* provides a memory-mapped register interface that applications can use to generate Instrumentation packets.

*The extension requirements are - ITM.*

**I<sub>BXWJ</sub>** The ITM is only available if the Main Extension is implemented.

*The extension requirements are - ITM.*

**R<sub>LMXS</sub>** The ITM generates Instrumentation packets, Synchronization packets, and the following protocol packets:

- Overflow packets.
- Local timestamp packets.
- Global timestamp packets.
- Extension packets.

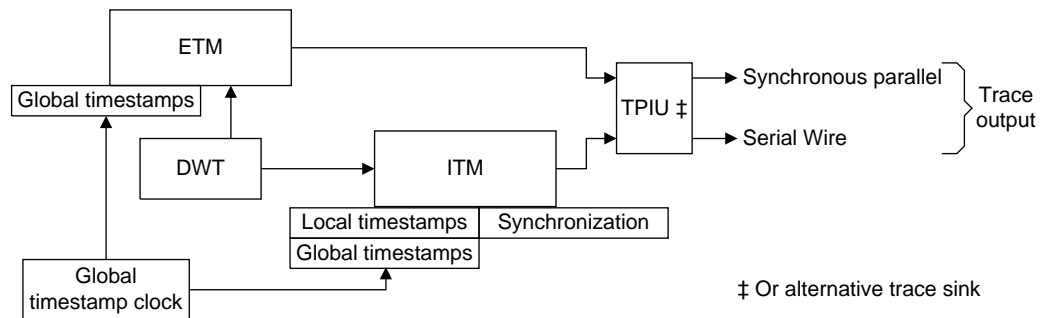
*The extension requirements are - ITM.*

**R<sub>XQRX</sub>** The ITM combines the following packets into a single trace stream:

- Instrumentation packets.
- Synchronization packets.
- Protocol packets.
- Hardware source packets that are generated by the DWT.

*The extension requirements are - ITM.*

**I<sub>FQLR</sub>** The following figure shows how the ITM relates to other debug components.



*The extension requirements are - ITM.*

**R<sub>BWJJ</sub>** When multiple sources are generating data at the same time, the ITM arbitrates using the following priorities:

**Synchronization, when required:** Priority level -1, highest.

**Instrumentation:** Priority level 0.

**Hardware source:** Priority level 1.

**Local timestamps:** Priority level 2.

**Global timestamp 1:** Priority level 3.

**Global timestamp 2:** Priority level 4.

*The extension requirements are - ITM.*

See also:

[Global timestamping.](#)

[B12.2 Data Watchpoint and Trace unit on page 278.](#)

[ITM and DWT Packet Protocol Specification.](#)

## B12.1.2 ITM operation

**R<sub>NKSC</sub>**

The ITM consists of:

- Up to 256 stimulus port registers, [ITM\\_STIMn](#).
- Up to eight enable registers, [ITM\\_TERN](#).
- An access control register, [ITM\\_TPR](#).
- A general control register, [ITM\\_TCR](#).

*The extension requirements are - ITM.*

**R<sub>MFDV</sub>**

The number of [ITM\\_STIMn](#) registers is an IMPLEMENTATION DEFINED multiple of eight. Software can discover the number of supported stimulus ports by writing all ones to the [ITM\\_TPR](#), and then reading how many bits are set to 1.

*The extension requirements are - ITM.*

**R<sub>CGVD</sub>**

If the ITM is disabled or not implemented, any Secure or Non-secure write to [ITM\\_STIMn](#) is ignored.

*The extension requirements are - ITM && S.*

**R<sub>NJTR</sub>**

Unprivileged and privileged software can always read all ITM registers.

*The extension requirements are - ITM.*

**R<sub>FFXF</sub>**

If the ITM is not implemented, the ITM registers are RAZ/WI.

*The extension requirements are - ITM.*

**R<sub>CSFV</sub>**

The [ITM\\_TPR](#) defines whether each group of eight [ITM\\_STIMn](#) registers, and their corresponding [ITM\\_TERN](#) bits, can be written by an unprivileged access.

*The extension requirements are - ITM.*

**R<sub>PTXV</sub>**

[ITM\\_STIMn](#) registers are 32-bit registers that support the following word-aligned accesses:

- Byte accesses, to access register bits[7:0].
- Halfword accesses, to access register bits[15:0].
- Word accesses, to access register bits[31:0].

*The extension requirements are - ITM.*

**R<sub>LNMW</sub>**

Non-word-aligned accesses are UNPREDICTABLE.

*The extension requirements are - ITM.*

**R<sub>NQVK</sub>**

[ITM\\_TCR.ITMENA](#) is a global enable bit for the ITM. A Cold reset clears this bit to 0, disabling the ITM.

*The extension requirements are - ITM.*

- R<sub>VRGP</sub>** The **ITM\_TERn** registers provide an enable bit for each stimulus port.  
*The extension requirements are - ITM.*
- R<sub>NTCR</sub>** When software writes to an enabled **ITM\_STIMn** register, the ITM combines the identity of the port, the size of the write access, and the data that is written, into an Instrumentation packet that it writes to a stimulus port output buffer. The ITM transmits packets from the output buffer to a trace sink.  
*The extension requirements are - ITM.*
- R<sub>XPRK</sub>** If an ITM implementation supports more than 32 stimulus ports, paging is used to indicate the stimulus port number. The stimulus port number is defined by the A field of the Instrumentation packet, byte 0 bits [7:3].  
*The extension requirements are - ITM.*
- R<sub>QBCH</sub>** An Extension packet is issued to define the page number, 0 to 7, with each subsequent Instrumentation packet.  
*The extension requirements are - ITM.*
- R<sub>TCRH</sub>** If **DEMCR.TRCENA** == 0 or **NoninvasiveDebugAllowed()** == FALSE, the ITM does not generate trace.  
*The extension requirements are - ITM.*
- R<sub>GRNM</sub>** The size of the stimulus port output buffer is IMPLEMENTATION DEFINED, but has at least one entry. The stimulus port output buffer is shared by all **ITM\_STIMn** registers.  
*The extension requirements are - ITM.*
- R<sub>SXNK</sub>** When the stimulus port output buffer is full, if software writes to any **ITM\_STIMn** register, the ITM discards the write data, and generates an Overflow packet.  
*The extension requirements are - ITM.*
- R<sub>SRPP</sub>** Reading the **ITM\_STIMn** register of any enabled stimulus port returns a value indicating the output buffer status and that the port is enabled.  
*The extension requirements are - ITM.*
- R<sub>XVVB</sub>** Reading an **ITM\_STIMn** register when the ITM is disabled, or when the individual stimulus port is disabled in the corresponding **ITM\_TERn** register, returns the value indicating that the output buffer cannot accept data because the port is disabled.  
*The extension requirements are - ITM.*
- R<sub>FXSL</sub>** Hardware source packets that are generated by any source use a separate output buffer. The output buffer status that is obtained by reading an **ITM\_STIMn** register is not affected by trace that is generated by any other source.  
*The extension requirements are - ITM && DWT-T.*
- R<sub>RGCV</sub>** Stalling is supported through an optional control, **ITM\_TCR.STALLENA**. When implemented and set to 1, the ITM can stall the PE to guarantee delivery of the following Hardware source packets:
- Data Trace Data Address.
  - Data Trace Data Value.
  - Data Trace Match.
  - Data Trace PC Value.
  - Exception Trace.
- The extension requirements are - ITM.*
- R<sub>NFJN</sub>** Stalling does not affect the DWT counters.  
*The extension requirements are - ITM && DWT-T.*

- R<sub>TNDP</sub>** The ITM might generate an Overflow packet while the PE is stalled, if the DWT generates:
- A Hardware source packet other than a Data trace packet or Exception packet.
  - A Data Trace PC value packet or Data Trace Match packet from a Cycle Counter comparator.
- The extension requirements are - ITM.*
- R<sub>CRKK</sub>** The ITM does not stall the PE in Secure state if `SecureHaltingDebugAllowed() == FALSE`.
- The extension requirements are - ITM && S.*
- R<sub>GRHW</sub>** The ITM does not stall the PE if `HaltingDebugAllowed() == FALSE`.
- The extension requirements are - ITM.*
- R<sub>BGCP</sub>** The ITM does not stall the PE in such a way as to deadlock the system.
- The extension requirements are - ITM.*
- R<sub>FRJG</sub>** The ITM does not stall the PE if the trace output is disabled.
- The extension requirements are - ITM.*
- R<sub>XRVL</sub>** The ITM does not stall for writes to the `ITM_STIMn` registers.
- The extension requirements are - ITM.*
- R<sub>HDLH</sub>** Instrumentation trace packets appear in the trace output in the order in which writes arrive at the `ITM_STIMn` registers.
- The extension requirements are - ITM.*
- R<sub>XNHX</sub>** It is IMPLEMENTATION DEFINED whether an ITM requires flushing of trace data to guarantee that data is output.
- The extension requirements are - ITM.*
- R<sub>TSXR</sub>** If periodic flushing is required, the ITM flushes trace data:
- When a Synchronization packet is generated.
  - When trace is disabled, meaning that either `DEMCR.TRCENA` is cleared to 0 or one or more of `ITM_TCR.{TXENA, SYNCENA, TSENA, SYNCENA}` is cleared to 0, and the buffered trace includes at least one corresponding packet type.
  - In response to other IMPLEMENTATION DEFINED flush requests from the system.
- The extension requirements are - ITM.*
- R<sub>MKFS</sub>** If a system supports multiple trace streams, the debugger writes a unique nonzero trace ID value to the `ITM_TCR.TraceBusID` field. The system uses this value to identify the individual trace streams. To avoid trace stream corruption, before modifying the `ITM_TCR.TraceBusID` a debugger does the following:
- It clears the `ITM_TCR.ITMENA` bit to 0, to disable the ITM.
  - It polls the `ITM_TCR.BUSY` bit until it returns to 0, indicating that the ITM is inactive.
- The extension requirements are - ITM.*

### B12.1.3 Timestamp support

- R<sub>RVLT</sub>** Timestamps provide information on the timing of event generation regarding their visibility at a trace output port.
- The extension requirements are - ITM.*
- R<sub>TFDG</sub>** An Armv8-M PE can implement either or both of the following types of timestamp:

- Local timestamps.
- Global timestamps.

The extension requirements are - *ITM*.

## Local timestamping

**R<sub>RMXM</sub>** Local timestamps provide delta timestamp values, meaning each local timestamp indicates the elapsed time since generating the previous local timestamp.

The extension requirements are - *ITM*.

**R<sub>WGBG</sub>** The ITM generates the local timestamps from the timestamp counter in the ITM unit.

The extension requirements are - *ITM*.

**R<sub>XLBH</sub>** The timestamp counter size is an IMPLEMENTATION DEFINED value that is less than or equal to 28 bits.

The extension requirements are - *ITM*.

**R<sub>GPXT</sub>** It is IMPLEMENTATION DEFINED whether the ITM supports synchronous clocking of the timestamp counter mode.

The extension requirements are - *ITM*.

**R<sub>SRJH</sub>** It is IMPLEMENTATION DEFINED whether the ITM and TPIU support asynchronous clocking of the timestamp counter mode.

The extension requirements are - *ITM*.

**R<sub>GHPG</sub>** [ITM\\_TCR.TSENA](#) enables Local timestamp packet generation.

The extension requirements are - *ITM*.

**R<sub>FSWG</sub>** When local timestamping is enabled and the DWT or ITM transfers a Hardware source or instrumentation trace packet to the appropriate output FIFO, and the timestamp counter is nonzero, the ITM:

- Generates a Local timestamp packet.
- Resets the timestamp counter to zero.

The extension requirements are - *ITM*.

**R<sub>BRRL</sub>** If the timestamp counter overflows, it continues counting from zero and the ITM generates an Overflow packet and transmits an associated Local timestamp packet at the earliest opportunity. If higher priority trace packets delay transmission of this Local timestamp packet, the timestamp packet has the appropriate nonzero local timestamp value.

The extension requirements are - *ITM*.

**R<sub>XFRH</sub>** The ITM can generate a Local timestamp packet relating to a single event packet, or to a stream of back-to-back packets if multiple events generate a packet stream without any idle time.

The extension requirements are - *ITM*.

**R<sub>QJJB</sub>** Local timestamp packets include status information that indicates any delay in one or both of:

- Transmission of the timestamp packet relative to the corresponding event packet.
- Transmission of the corresponding event packet relative to the event itself.

The extension requirements are - *ITM*.

**R<sub>NDCK</sub>** If the ITM cannot generate a Local timestamp packet synchronously with the corresponding event packet, the timestamp count continues to increment until the ITM can generate a Local timestamp packet.

The extension requirements are - *ITM*.

**R<sub>TBMX</sub>** The ITM compresses the count value in the timestamp packet by removing leading zeroes, and transmits the smallest packet that can hold the required count value.

*The extension requirements are - ITM.*

**I<sub>SQLG</sub>** To prevent overflow, Arm recommends that the ITM emits a Local timestamp packet before the timestamp counter overflows.

*The extension requirements are - ITM.*

### Local timestamp clocking options

**R<sub>DSTG</sub>** If the implementation supports both synchronous and asynchronous clocking of the local timestamp counter, [ITM\\_TCR.SWOENA](#) selects the clocking mode.

*The extension requirements are - ITM.*

**R<sub>BDWS</sub>** When software selects synchronous clocking, when local timestamping is enabled, the PE clock drives the timestamp counter, and the counter increments on each PE clock cycle.

*The extension requirements are - ITM.*

**I<sub>JQJD</sub>** When software selects synchronous clocking, whether local timestamps are generated in Debug state is IMPLEMENTATION DEFINED. Arm recommends that entering Debug state disables local timestamping, regardless of the value of the [ITM\\_TCR.TSENA](#) bit.

*The extension requirements are - ITM.*

**R<sub>JDRD</sub>** When software selects asynchronous clocking, and enables local timestamping, the TPIU output interface clock drives the timestamp counter, through a configurable prescaler. The rate of asynchronous clocking depends on the output encoding scheme. This clock might be asynchronous to the PE clock.

*The extension requirements are - ITM.*

**R<sub>NGDW</sub>** When asynchronous clocking is implemented, whether the incoming clock signal can be divided before driving the local timestamping counter is IMPLEMENTATION DEFINED.

*The extension requirements are - ITM.*

**R<sub>RMTN</sub>** If the implementation supports division of the incoming asynchronous clock signal, [ITM\\_TCR.TSPrescale](#) sets the prescaler divide value.

*The extension requirements are - ITM.*

**R<sub>SKCP</sub>** Software only selects asynchronous clocking when the TPIU is configured to use an output mode that supports asynchronous clocking.

*The extension requirements are - ITM && TPIU.*

**R<sub>JGCF</sub>** When software selects asynchronous clocking and the TPIU asynchronous interface is idle, the ITM holds the timestamp counter at zero. This means that the ITM does not generate a local timestamp on the first packet after an idle on the asynchronous interface.

*The extension requirements are - ITM && TPIU.*

See also:

[B12.4 Trace Port Interface Unit on page 299.](#)

## Global timestamping

**I<sub>DKSD</sub>** Global timestamps provide absolute timestamp values, which are based on a system global timestamp clock. They provide synchronization between different trace sources in the system.

*The extension requirements are - ITM.*

**R<sub>HBWD</sub>** If an implementation includes Global timestamping, the ITM generates *Global timestamp* (GTS) packets, which are based on a global timestamp clock.

*The extension requirements are - ITM.*

**R<sub>KWQJ</sub>** The size of the global timestamp is either 48 bits or 64 bits. The choice between these two options is IMPLEMENTATION DEFINED.

*The extension requirements are - ITM.*

**R<sub>SRDF</sub>** To transfer the global timestamp, two formats of Global timestamp packets are defined:

- The first packet format, Global timestamp 1 packet, holds the value of the least significant timestamp bits[25:0], and wrap and clock change indicators.
- The second packet format, Global timestamp 2 packet, holds the value of the high-order timestamp bits:
  - Bits[47:26], if a 48-bit global timestamp is supported.
  - Bits[63:26], if a 64-bit global timestamp is supported.

*The extension requirements are - ITM.*

**R<sub>VGBT</sub>** The ITM generates a full Global timestamp packet, consisting of Global timestamp 1 packet Global timestamp 2 packet, in the following circumstances:

- When software first enables global timestamps, by changing the value of the [ITM\\_TCR.GTSFREQ](#) field from zero to a nonzero value.
- When the system asserts the clock ratio change signal in the external ITM timestamp interface.
- In response to a Synchronization packet request, even if [ITM\\_TCR.SYNCENA](#) == 0.
- When the ITM has to generate a global timestamp, and the ITM detects that the value of the high-order bits of the global timestamp have changed since the Global timestamp 2 packet was last generated.

*The extension requirements are - ITM.*

**R<sub>XQWL</sub>** If the global timestamp generated by the ITM does not have to be a full global timestamp, the ITM generates only a single Global timestamp 1 packet.

*The extension requirements are - ITM.*

**R<sub>DJLN</sub>** When the ITM generates a global timestamp, it does so after a non-delayed Instrumentation or Hardware Source packet. The Global Timestamp 1 packet is always associated with the most recently output non-delayed Instrumentation or Hardware Source packet.

*The extension requirements are - ITM.*

**R<sub>WDCX</sub>** When the ITM generates a full global timestamp:

1. The ITM first generates the Global timestamp 1 packet with timestamp bits[25:0], with the applicable bit of the Wrap and ClockCh bits in that packet set to 1 to indicate that the high-order bits of the timestamp will also be output. This is the packet that the ITM outputs immediately after a non-delayed trace packet.
2. Because of packet prioritization, the ITM might have to transmit other trace packets before it can output the Global timestamp 2 packet that contains the high-order bits of the timestamp. It might also have to transmit another Global timestamp packet. If so, it outputs the Global timestamp 1 packet with timestamp bits[25:0] and the Wrap bit set to 1.

3. The ITM later generates the Global timestamp 2 packet with the high-order timestamp bits for the most recently transmitted Global timestamp 1 packet.

The extension requirements are - *ITM*.

See also:

[B12.1.4 Synchronization support](#) .

[B12.1.5 Continuation bits](#) .

[ITM and DWT Packet Protocol Specification](#).

### B12.1.4 Synchronization support

**I<sub>LRJT</sub>** An external debugger uses Synchronization Packets to recover bit-to-byte alignment information in a serial data stream.

The extension requirements are - *ITM*.

**I<sub>LVGD</sub>** Synchronization packets are independent of timestamp packets.

The extension requirements are - *ITM*.

**R<sub>QRFL</sub>** If the ITM connects to a parallel trace port interface, it must generate periodic Synchronization packets. An ITM connected to an asynchronous serial trace port interface can generate Synchronization packets

The extension requirements are - *ITM*.

**I<sub>JNJV</sub>** Arm recommends that software disables Synchronization packets when using an asynchronous serial trace port, to reduce the data stream bandwidth.

The extension requirements are - *ITM*.

**R<sub>RMND</sub>** If [ITM\\_TCR.SYNCENA](#) == 1, the ITM outputs a Synchronization packet:

- When it is first enabled.
- If [DWT\\_CYCCNT](#) is implemented and [DWT\\_CTRL.SYNCTAP](#) is nonzero, in response to a Synchronization packet request from the DWT unit.
- If [TPIU\\_PSCR](#) is implemented, in response to a Synchronization packet request from the TPIU:
  - If [DWT\\_CYCCNT](#) is not implemented, [TPIU\\_PSCR](#) is implemented.
  - If [DWT\\_CYCCNT](#) is implemented, it is IMPLEMENTATION DEFINED whether [TPIU\\_PSCR](#) is implemented.
- In response to other IMPLEMENTATION DEFINED Synchronization packet requests from the system.
- On exit from Debug state.

The extension requirements are - *ITM*. Note, might require additional extensions as described in the rule.

See also:

[DWT\\_CTRL.SYNCTAP](#).

### B12.1.5 Continuation bits



Chapter B12. Debug and Trace Components

B12.1. Instrumentation Trace Macrocell

**I<sub>BFBMX</sub>** A Synchronization packet consists of a bit stream of at least 47 zero bits followed by a one bit. The final bit is the byte alignment marker, and therefore bit[7] of the last byte of a Synchronization packet is always one.

*The extension requirements are - ITM. Note, When the ITM supports more than 32 Stimulus ports.*

**R<sub>JNVH</sub>** The longest Extension packet is always 5 bytes. In an Extension packet, bit[7] of each byte, including the header byte, but not including the last byte of a 5-byte packet, is a continuation bit, C. Bit[7] of the last byte of a 5-byte Extension packet is part of the extension field. Bit[7] of the last byte of a fewer-than-5-byte Extension packet is always zero.

*The extension requirements are - ITM.*

**R<sub>XFTL</sub>** For all other protocol packets, bit[7] of each byte, including the header byte, but not including the last byte of a 7-byte packet, is a continuation bit, C. Bit[7] of the last byte of a packet is always zero.

*The extension requirements are - ITM.*

**R<sub>BBSF</sub>** Each packet type defines its maximum packet length. Except for Global timestamp 2 and Synchronization packets, the longest defined packet is 5 bytes.

*The extension requirements are - ITM.*

**R<sub>DPJG</sub>** The continuation bit, C, is defined as:

**0:** This is the last byte of the packet.

**1:** This is not the last byte of the packet.

*The extension requirements are - ITM.*

## B12.2 Data Watchpoint and Trace unit

### B12.2.1 About the DWT

**R<sub>QOLQ</sub>**

The *Data Watchpoint and Trace* (DWT) unit provides the following features:

- Comparators that support:
  - Use as a single comparator for instruction address matching or data address matching.
  - Use in linked pairs for instruction address range matching or data address range matching.
- Generation, on a comparator match, of:
  - A debug event that causes the PE either to enter Debug state or, if the Main Extension is implemented, to take a DebugMonitor exception.
  - Signaling a match to an ETM, if implemented.
  - Signaling a match to another external resource.
- External instruction address sampling using an instruction address sample register.

*The extension requirements are - DWT-T && (DebugMonitor exception || Halting debug). Note, some comparator matches require ETM.*

**R<sub>KBMX</sub>**

If the Main Extension is implemented, the DWT provides the following features:

- An optional cycle counter.
- Comparators that support:
  - Use as a single comparator for cycle counter matching, if the cycle counter is implemented.
  - Use as a single comparator for data value matching.
  - Use in linked pairs for data value matching at a specific data address.

*The extension requirements are - DWT-T && M.*

**R<sub>DVJV</sub>**

If the Main Extension and the ITM are implemented, the DWT provides the following trace generation features:

- Generating one or more trace packets on a comparator match.
- Generating periodic trace packets for software profiling.
- Exception trace.
- Performance profiling counters that generate trace.

*The extension requirements are - DWT-T && M && ITM.*

**R<sub>CPXJ</sub>**

If **DWT\_CTRL.NOTRCPKT** is 1, there is no DWT trace support.

*The extension requirements are - DWT-T.*

**R<sub>FKFP</sub>**

If **DWT\_CTRL.NOCYCCNT** is 1, there is no cycle counter support.

*The extension requirements are - DWT-T.*

**R<sub>BKGF</sub>**

If **DWT\_CTRL.NOPRFCNT** is 1, there is no profiling counter support.

*The extension requirements are - DWT-T.*

**R<sub>HFTT</sub>**

The **DWT\_CTRL.NUMCOMP** field indicates the number of implemented DWT comparators, which is in the range 0-15.

*The extension requirements are - DWT-T.*

**R<sub>WOLX</sub>**

If the Main Extension is not implemented, Cycle counter, Data value, Linked data value, and Data address with

value comparators and all trace features are not supported.

The extension requirements are - **M** && **DWT-T**.

**R<sub>SSWT</sub>** Data trace packets are only generated for comparators 0-3.

The extension requirements are - **DWT-T**.

**R<sub>CRHX</sub>** When a DWT implementation includes one or more comparators, which comparator features are supported, and by which comparators, is IMPLEMENTATION DEFINED.

The extension requirements are - **DWT-T**.

## B12.2.2 DWT unit operation

**I<sub>WTSS</sub>** For each implemented comparator, a set of registers defines the comparator operation. For comparator *n*:

- **DWT\_COMP<sub>n</sub>** holds a value for the comparison.
- **DWT\_FUNCTION<sub>n</sub>** defines the operation of the comparator.

The extension requirements are - **DWT-T**.

**R<sub>XBRD</sub>** A *Secure match* is a match that is generated by one of the following:

- Vector fetches where NS-Req has a value of Secure for the operation.
- The hardware stacking or unstacking of registers, where NS-Req has a value of Secure for the operation, on any of:
  - Exception entry.
  - Exception exit.
  - Function call entry.
  - Function return.
  - Lazy state preservation.
- An operation that is generated by an instruction that is executed in Secure state, including:
  - An Instruction address match for an instruction that is executed in Secure state.
  - A Data address or Data value match for a load or store that is generated by an instruction that is executed in Secure state.

The extension requirements are - **DWT-T** && **S**.

**R<sub>DVCN</sub>** A Secure match can be generated by a cycle counter match in Secure state if **DWT\_CTRL.CYCDISS** == 1.

The extension requirements are - **DWT-T** && **S**.

**R<sub>MGGT</sub>** For a comparator <*n*>, all matches are prohibited if one or more of the following conditions apply:

- **DEMCR.TRCENA** == 0 or **NoninvasiveDebugAllowed()** == FALSE.
- **DWT\_FUNCTION.ACTION** specifies a debug event and all the following conditions apply:
  - **HaltingDebugAllowed()** == FALSE or **DHCSR.C\_DEBUGEN** == 0.
  - The Main Extension is not implemented or **DEMCR.MON\_EN** == 0.

The extension requirements are - **DWT-T**.

**R<sub>GFLN</sub>** Secure matches are prohibited for a comparator if one of the following conditions applies:

- **DWT\_FUNCTION.ACTION** specifies a trace or trigger event and **SecureNoninvasiveDebugAllowed()** == FALSE.

- **DWT\_FUNCTION.ACTION** specifies a debug event and all of the following conditions apply:
  - **DHCSR.S\_SDE** == 0.
  - The Main Extension is not implemented or **DEMCR.SDME** == 0.

*The extension requirements are - DWT-T && S. Note, M required if DEMCR.SDME == 1.*

|                         |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                      |
|-------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>R<sub>HCFP</sub></b> | For address and value comparisons, the control register values and the current execution priority and Security state relate to the state of the PE when it generated the transaction that is being matched against.<br><br><i>The extension requirements are - DWT-T &amp;&amp; S.</i>                                                                                                                                                                                                                                                               |
| <b>R<sub>FFKV</sub></b> | Between a change to the debug authentication interface, <b>DHCSR</b> or <b>DEMCR</b> , that disables debug and a following <b>context synchronization event</b> , it is UNPREDICTABLE whether the DWT uses the old values or the new values.<br><br><i>The extension requirements are - DWT-T.</i>                                                                                                                                                                                                                                                   |
| <b>R<sub>VTNJ</sub></b> | Where the DWT operation rules prohibit a match being generated, a match is not generated, even if the programmers' model defines it as being UNPREDICTABLE whether a comparator generates a match as the result of the way in which the DWT is programmed.<br><br><i>The extension requirements are - DWT-T.</i>                                                                                                                                                                                                                                     |
| <b>R<sub>PKRK</sub></b> | If <b>DEMCR.TRCENA</b> == 0 or <code>NoninvasiveDebugAllowed()</code> == FALSE, <b>DWT_CTRL.FOLDEVTENA</b> , <b>LSUEVTENA</b> , <b>SLEEPEVTENA</b> , <b>EXCEVTENA</b> , and <b>CPIEVTENA</b> are ignored, and these fields have an <b>Effective value</b> of 0.<br><br><i>The extension requirements are - DWT-T.</i>                                                                                                                                                                                                                                |
| <b>R<sub>GDMN</sub></b> | If <b>DEMCR.TRCENA</b> == 0 or <code>NoninvasiveDebugAllowed()</code> == FALSE, the DWT does not generate any trace packets.<br><br><i>The extension requirements are - DWT-T.</i>                                                                                                                                                                                                                                                                                                                                                                   |
| <b>R<sub>FHWV</sub></b> | If <code>SecureNoninvasiveDebugAllowed()</code> == FALSE, <b>DWT_CTRL.FOLDEVTENA</b> , <b>LSUEVTENA</b> , <b>SLEEPEVTENA</b> , <b>EXCEVTENA</b> , and <b>CPIEVTENA</b> are ignored and these fields have an <b>Effective value</b> of 0 in Secure state.<br><br><i>The extension requirements are - DWT-T &amp;&amp; S.</i>                                                                                                                                                                                                                          |
| <b>R<sub>WSRR</sub></b> | If <code>SecureNoninvasiveDebugAllowed()</code> == FALSE, Exception trace packets are not generated if the exception number in the packet represents a Secure exception: <ul style="list-style-type: none"><li>• Exception entry packets are not generated for exceptions that are taken to Secure state.</li><li>• Exception exit packets are not generated for exits from Secure state.</li><li>• Exception return packets are not generated for returns to Secure state.</li></ul><br><i>The extension requirements are - DWT-T &amp;&amp; S.</i> |
| <b>R<sub>DFWR</sub></b> | Exception trace packets appear in the same order as for a simple sequential execution of the exception handling.<br><br><i>The extension requirements are - DWT-T.</i>                                                                                                                                                                                                                                                                                                                                                                               |
| <b>R<sub>XDVS</sub></b> | The cycle counter, <b>DWT_CYCCNT</b> , and the <b>POSTCNT</b> counter are disabled when <b>DEMCR.TRCENA</b> == 0, but are not otherwise affected by debug authentication.<br><br><i>The extension requirements are - DWT-T.</i>                                                                                                                                                                                                                                                                                                                      |
| <b>R<sub>RTJR</sub></b> | The cycle counter does not count in Secure state when <b>DWT_CTRL.CYCDISS</b> is set to 1. This is independent of Secure debug authentication.<br><br><i>The extension requirements are - DWT-T &amp;&amp; S.</i>                                                                                                                                                                                                                                                                                                                                    |

**R<sub>BRSR</sub>** When the DWT generates a match, **DWT\_FUNCTION.MATCHED** is set to 1, unless the comparator is a Data address limit or Instruction address limit comparator, in which case **DWT\_FUNCTION.MATCHED** is UNKNOWN.

*The extension requirements are - DWT-T.*

**R<sub>NRGV</sub>** When the DWT generates a match, then if **DWT\_FUNCTION.ACTION** specifies a debug event, then **DHCSR.C\_HALT** is set to 1 if all of the following conditions are true:

- **HaltingDebugAllowed()** == TRUE.
- **DHCSR.C\_DEBUGEN** == 1.
- **DHCSR.S\_HALT** == 0.
- Either the match is not a Secure match or **DHCSR.S\_SDE** == 1.

*The extension requirements are - DWT-T.*

**R<sub>PJGW</sub>** When the DWT generates a match, then if **DWT\_FUNCTION.ACTION** specifies a debug event, **DEMCR.MON\_PEND** is set to 1 if all of the following conditions apply:

- **HaltingDebugAllowed()** == FALSE, **DHCSR.C\_DEBUGEN** == 0, or the match is a Secure match and **DHCSR.S\_SDE** == 0.
- **DEMCR.MON\_EN** == 1.
- Either the DebugMonitor exception group priority is greater than the execution priority of the access and the watchpoint was not generated by a lazy state preservation access, or **FPCCR.MONRDY** has a value of 1 and the watchpoint was generated by lazy state preservation.

*The extension requirements are - DWT-T && M.*

**R<sub>FTBG</sub>** When the DWT generates a match, then a Data trace match packet is generated, if all of the following conditions apply:

- **SecureNoninvasiveDebugAllowed()** == FALSE.
- **DWT\_FUNCTION.ACTION** specifies generating a Data trace PC value packet.
- The instruction address that would be included in the packet refers to an instruction that was executed in Secure state.

Otherwise, the type of trace packet that is specified by **DWT\_FUNCTION.ACTION** is generated.

*The extension requirements are - DWT-T && M && S.*

**R<sub>FNDW</sub>** An access that results in a MemManage fault or SecureFault exception because of the alignment, SAU, IDAU, or MPU checks, is not observed by the DWT, and cannot generate a match.

*The extension requirements are - DWT-T && (S || M && MPU).*

**R<sub>PJJB</sub>** The DWT treats hardware accesses to the stack as data accesses:

- For registers pushed to the stack by hardware as part of an exception entry or lazy state preservation.
- For registers popped from the stack by hardware as part of an exception return.

*The extension requirements are - DWT-T.*

**R<sub>NONR</sub>** The DWT treats hardware accesses to the stack as data accesses:

- For registers pushed to the stack by hardware as part of a Non-secure function call.
- For registers popped from the stack by hardware as part of a Non-secure function.

*The extension requirements are - DWT-T && S.*

**R<sub>SFSC</sub>** Where a hardware access to the stack generates a Data trace PC value packet, the PC value in the packet will be as follows:

- On exception entry or a function call, the PC value will be the return address for the exception or function

call.

- On lazy state preservation the PC value is the address of the instruction that triggered the lazy state preservation.
- On exception return or Non-secure function return the PC value is either:
  - The address of the instruction that caused the exception return or the Non-secure function return.
  - The `EXC_RETURN` or `FNC_RETURN` payload value used in the exception return or the Non-secure function return.

*The extension requirements are - DWT-T.*

### B12.2.3 Constraints on programming DWT comparators

**R<sub>MSPS</sub>**

If a DWT comparator, <n>, or pair of comparators, <n> and <n+1>, is programmed with a reserved combination of `DWT_FUNCTION.MATCH` and `DWT_FUNCTION.ACTION`, then it is UNPREDICTABLE whether any comparator:

- Behaves as if disabled.
- Generates a match, setting `DWT_FUNCTION.MATCHED` bit to an UNKNOWN value, and any of the following:
  - Asserts `CMPMATCH`.
  - Generates a debug event.
  - Generates one or more trace packets.

*The extension requirements are - DWT-T.*

**R<sub>GPLQ</sub>**

Combinations of `DWT_FUNCTION.MATCH` and `DWT_FUNCTION.ACTION` that are not specified as valid combinations are reserved.

*The extension requirements are - DWT-T.*

**R<sub>JHZK</sub>**

It is IMPLEMENTATION DEFINED which values of `DWT_FUNCTION.MATCH` are valid for counter <n>. `DWT_FUNCTION.ID` defines which values are valid. Values that are not valid are reserved.

*The extension requirements are - DWT-T.*

**R<sub>CNHN</sub>**

The valid combinations of `DWT_FUNCTION.MATCH` and `DWT_FUNCTION.ACTION` for a single comparator, and the events and Data trace packets that the comparator can generate from matching a single access, are identified in the following table.

In the table:

-: means that the packet or event is not generated.

**Yes:** means that the packet or event is generated on a comparator match.

| Comparator Type         | MATCH                     | ACTION | Debug Event | Data Trace   |                       |                     |                   |
|-------------------------|---------------------------|--------|-------------|--------------|-----------------------|---------------------|-------------------|
|                         |                           |        |             | Match Packet | PC Value Match Packet | Data Address Packet | Data Value Packet |
| Disabled                | 0b0000                    | 0bxx   | -           | -            | -                     | -                   | -                 |
| Cycle Counter           | 0b0001                    | 0b00   | -           | -            | -                     | -                   | -                 |
|                         |                           | 0b01   | Yes         | -            | -                     | -                   | -                 |
|                         |                           | 0b10   | -           | Yes          | -                     | -                   | -                 |
|                         |                           | 0b11   | -           | -            | Yes                   | -                   | -                 |
| Instruction Address     | 0b0010                    | 0b00   | -           | -            | -                     | -                   | -                 |
|                         |                           | 0b01   | Yes         | -            | -                     | -                   | -                 |
|                         |                           | 0b10   | -           | Yes          | -                     | -                   | -                 |
| Data address            | 0b01xx<br>(not<br>0b0111) | 0b00   | -           | -            | -                     | -                   | -                 |
|                         |                           | 0b01   | Yes         | -            | -                     | -                   | -                 |
|                         |                           | 0b10   | -           | Yes          | -                     | -                   | -                 |
|                         |                           | 0b11   | -           | -            | Yes                   | -                   | -                 |
| Data value              | 0b10xx<br>(not<br>0b1011) | 0b00   | -           | -            | -                     | -                   | -                 |
|                         |                           | 0b01   | Yes         | -            | -                     | -                   | -                 |
|                         |                           | 0b10   | -           | Yes          | -                     | -                   | -                 |
|                         |                           | 0b11   | -           | -            | Yes                   | -                   | -                 |
| Data address with value | 0b11xx (not<br>0b1111)    | 0b10   | -           | -            | -                     | -                   | Yes               |
|                         |                           | 0b11   | -           | -            | Yes                   | -                   | Yes               |

The extension requirements are - *DWT-T*. Note, Cycle counter, Data value and Data address with value are only available if *M* is implemented.

### Instruction address range

R<sub>DKHG</sub>

To match an instruction that is in an instruction address range, the following conditions are met:

- The first comparator, <n-1>, is programmed for *Instruction address*.
- The second comparator, <n>, is programmed for *Instruction address limit*.

The extension requirements are - *DWT-T*.

R<sub>LNQD</sub>

The valid combinations of [DWT\\_FUNCTION.MATCH](#) and [DWT\\_FUNCTION.ACTION](#) for an instruction address range, and the events and data trace packets that matching a single access can generate, are specified in the following table.

In the table:

-: means that the packet or event is not generated.

**First:** means that the packet or event is generated by the first comparator match.

**Second:** means that the packet or event is generated by the second comparator match.

| MATCH  |        | ACTION |      | Data Trace  |              |                 |                     |                   |
|--------|--------|--------|------|-------------|--------------|-----------------|---------------------|-------------------|
| <n-1>  | <n>    | <n-1>  | <n>  | Debug Event | Match packet | PC Value packet | Data Address packet | Data Value packet |
| 0b0000 | 0b0011 | 0bxx   | 0bxx | -           | -            | -               | -                   | -                 |
| 0b0010 | 0b0011 | 0b00   | 0b00 | -           | -            | -               | -                   | -                 |
|        |        | 0b00   | 0b11 | -           | -            | Second          | -                   | -                 |
|        |        | 0b01   | 0b00 | First       | -            | -               | -                   | -                 |
|        |        | 0b10   | 0b00 | -           | First        | -               | -                   | -                 |

The extension requirements are - *DWT-T* && *M*.

R<sub>VDRJ</sub>

If the Main Extension is not implemented the valid combinations of [DWT\\_FUNCTION.MATCH](#) and [DWT\\_FUNCTION.ACTION](#) for an instruction address range, and the events and data trace packets that matching

a single access can generate, are specified in the following table.

In the table:

-: means that the packet or event is not generated.

**First:** means that the packet or event is generated by the first comparator match.

**Second:** means that the packet or event is generated by the second comparator match.

| MATCH  |        | ACTION |      | Data Trace  |              |                 |                     |                   |
|--------|--------|--------|------|-------------|--------------|-----------------|---------------------|-------------------|
| <n-1>  | <n>    | <n-1>  | <n>  | Debug Event | Match packet | PC Value packet | Data Address packet | Data Value packet |
| 0b0000 | 0b0011 | 0bxx   | 0bxx | -           | -            | -               | -                   | -                 |
| 0b0010 | 0b0011 | 0b00   | 0b00 | -           | -            | -               | -                   | -                 |
|        |        | 0b01   | 0b00 | First       | -            | -               | -                   | -                 |

The extension requirements are - *DWT-T* && *!M*.

### Data address range

**R<sub>LDGR</sub>**

To match a data access in a data address range, the following conditions are met:

- The first comparator, <n-1>, is programmed for either *Data address* or *Data address with value*.
- The second comparator, <n>, is programmed for *Data address limit*.

The extension requirements are - *DWT-T* && *M*.

**R<sub>PSBJ</sub>**

The valid combinations of [DWT\\_FUNCTION.MATCH](#) and [DWT\\_FUNCTION.ACTION](#) for a data address range, and the events and data trace packets that matching a single access can generate, are specified in the following table.

In the table:

-: means that the packet or event is not generated.

**First:** means that the packet or event is generated by the first comparator match.

**Second:** means that the packet or event is generated by the second comparator match.

| MATCH   |        | ACTION |      | Data Trace  |              |                 |                     |                   |
|---------|--------|--------|------|-------------|--------------|-----------------|---------------------|-------------------|
| <n-1>   | <n>    | <n-1>  | <n>  | Debug Event | Match packet | PC Value packet | Data Address packet | Data Value packet |
| 0b0000  | 0b0111 | 0bxx   | 0bxx | -           | -            | -               | -                   | -                 |
| 0b01xx  | 0b0111 | 0b00   | 0b00 | -           | -            | -               | -                   | -                 |
| (not    |        | 0b00   | 0b11 | -           | -            | -               | Second              | -                 |
| 0b0111) |        | 0b01   | 0b00 | First       | -            | -               | -                   | -                 |
|         |        | 0b10   | 0b00 | -           | First        | -               | -                   | -                 |
|         |        | 0b11   | 0b00 | -           | -            | First           | -                   | -                 |
|         |        | 0b11   | 0b11 | -           | -            | First           | Second              | -                 |
| 0b11xx  | 0b0111 | 0b10   | 0b00 | -           | -            | -               | -                   | First             |
| (not    |        | 0b10   | 0b11 | -           | -            | -               | Second              | First             |
| 0b1111) |        | 0b10   | 0b11 | -           | -            | First           | -                   | First             |
|         |        | 0b11   | 0b11 | -           | -            | First           | Second              | First             |

The extension requirements are - *DWT-T*.

**R<sub>HDMX</sub>**

If the Main Extension is not implemented the valid combinations of and for a data address range, and the events and data trace packets that matching a single access can generate, are specified in the following table.

In the table:

-: means that the packet or event is not generated.



**First:** means that the packet or event is generated by the first comparator match.

**Second:** means that the packet or event is generated by the second comparator match.

| MATCH   |        | ACTION |      | Data Trace  |              |                 |                     |                   |
|---------|--------|--------|------|-------------|--------------|-----------------|---------------------|-------------------|
| <n-1>   | <n>    | <n-1>  | <n>  | Debug Event | Match packet | PC Value packet | Data Address packet | Data Value packet |
| 0b0000  | 0b0111 | 0bxx   | 0bxx | -           | -            | -               | -                   | -                 |
| 0b01xx  | 0b0111 | 0b00   | 0b00 | -           | -            | -               | -                   | -                 |
| (not    |        | 0b00   | 0b11 | -           | -            | -               | Second              | -                 |
| 0b0111) |        | 0b01   | 0b00 | First       | -            | -               | -                   | -                 |

The extension requirements are - *DWT-T* && *!M*.

### Data value at specific address

**R<sub>KFHV</sub>** Matching data values at specific data addresses is possible only if the Main Extension is implemented.

The extension requirements are - *DWT-T*.

**R<sub>NNXD</sub>** To match a data value at a specific data address, the following conditions are met:

- The first comparator, <n-1>, is programmed for either *Data address* or *Data address with value*.
- The second comparator, <n>, is programmed for *Linked data value*.

The extension requirements are - *DWT-T*.

**R<sub>JKGJ</sub>** The first comparator matches any access that matches the address. The second matches only accesses that match the address and the data value.

The extension requirements are - *DWT-T*.

**R<sub>NTSD</sub>** The valid combinations of [DWT\\_FUNCTION.MATCH](#) and [DWT\\_FUNCTION.ACTION](#) for a linked data value, and the events and data trace packets that matching a single access can generate, are specified in the following table.

In the table:

-: means that the packet or event is not generated.

**First:** means that the packet or event is generated by the first comparator match.

**Second:** means that the packet or event is generated by the second comparator match.

**Both:** means that a first packet is generated by a first comparator match, even if the Linked data value comparator does not match, and a second packet is generated by the second comparator match, if both comparators match.

| MATCH           |        | ACTION |      | Data Trace  |              |                 |                     |                   |
|-----------------|--------|--------|------|-------------|--------------|-----------------|---------------------|-------------------|
| <n-1>           | <n>    | <n-1>  | <n>  | Debug Event | Match packet | PC Value packet | Data Address packet | Data Value packet |
| 0b0000          | 0b1011 | 0bxx   | 0bxx | -           | -            | -               | -                   | -                 |
| (not<br>0b0111) | 0b1011 | 0b00   | 0b00 | -           | -            | -               | -                   | -                 |
|                 |        | 0b00   | 0b01 | Second      | -            | -               | -                   | -                 |
|                 |        | 0b00   | 0b10 | -           | Second       | -               | -                   | -                 |
|                 |        | 0b01   | 0b00 | First       | -            | -               | -                   | -                 |
|                 |        | 0b01   | 0b10 | First       | Second       | -               | -                   | -                 |
|                 |        | 0b10   | 0b00 | -           | First        | -               | -                   | -                 |
|                 |        | 0b10   | 0b01 | Second      | First        | -               | -                   | -                 |
|                 |        | 0b11   | 0b00 | -           | -            | First           | -                   | -                 |
|                 |        | 0b11   | 0b01 | Second      | -            | First           | -                   | -                 |
|                 |        | 0b11   | 0b10 | -           | Second       | First           | -                   | -                 |
| (not<br>0b1111) | 0b1011 | 0b10   | 0b00 | -           | -            | -               | -                   | First             |
|                 |        | 0b10   | 0b01 | Second      | -            | -               | -                   | First             |
|                 |        | 0b10   | 0b10 | -           | Second       | -               | -                   | First             |
|                 |        | 0b11   | 0b00 | -           | -            | First           | -                   | First             |
|                 |        | 0b11   | 0b01 | Second      | -            | First           | -                   | First             |
|                 |        | 0b11   | 0b10 | -           | Second       | First           | -                   | First             |

The extension requirements are - DWT-T.

### B12.2.4 CMPMATCH trigger events

**I<sub>VNCC</sub>** The **CMPMATCH** events signal watchpoint matches.

The extension requirements are - DWT-T.

**R<sub>PRJG</sub>** The implementation of **CMPMATCH** is IMPLEMENTATION DEFINED.

The extension requirements are - DWT-T.

**R<sub>FTWC</sub>** If an ETM is implemented, **CMPMATCH** events are output to the ETM.

The extension requirements are - DWT-T && ETM.

**R<sub>TMZX</sub>** If an ETM is not implemented, the effect of **CMPMATCH** is IMPLEMENTATION DEFINED, including whether the trigger event has any observable effect or whether observable effects are visible to other components in the system.

The extension requirements are - DWT-T.

**R<sub>XXKM</sub>** For all enabled watchpoints, if **DWT\_FUNCTION<sub>n</sub>** is not programmed as an Instruction address limit comparator and is not programmed as a Data address limit comparator, **CMPMATCH[n]** is triggered on a comparator match.

The extension requirements are - DWT-T.

**R<sub>GVHS</sub>** For all enabled watchpoints, if **DWT\_FUNCTION<sub>n</sub>** is programmed as an Instruction address limit or Data address limit comparator, it is UNPREDICTABLE whether **CMPMATCH[n]** is triggered on a comparator match.

The extension requirements are - DWT-T.

### B12.2.5 Matching in detail

#### Instruction address matching in detail

**R<sub>GNVB</sub>** The DWT checks all instructions that are executed by a simple sequential execution of the program and do not

generate any exception for an instruction address match, including conditional instructions that fail their condition code check.

*The extension requirements are - DWT-T.*

**R<sub>NOGR</sub>**

An instruction might be checked by the DWT for an instruction address match if it either:

- Is executed by a simple sequential execution of the program and generates a synchronous exception.
- Would be executed by the sequential execution of the program but is abandoned because of an asynchronous exception.

*The extension requirements are - DWT-T.*

**R<sub>KJJC</sub>**

Speculative instruction prefetches, other than those that would be executed by the sequential execution of the program but that are abandoned because of asynchronous exceptions, do not generate matches.

*The extension requirements are - DWT-T.*

**R<sub>DSDT</sub>**

For all instruction address matches, if bit[0] of the comparator address has a value of 1, it is UNPREDICTABLE whether a match is generated when the other address bits match.

*The extension requirements are - DWT-T.*

**R<sub>KLXM</sub>**

For single instruction address matches, an instruction matches if the address of the first byte of the instruction matches the comparator address.

*The extension requirements are - DWT-T.*

**R<sub>FXFM</sub>**

For single address matches, if the instruction at address A is a 4-byte T32 instruction, and the address A+2 matches but the address A does not match, it is UNPREDICTABLE whether a match is generated.

*The extension requirements are - DWT-T.*

**R<sub>DNKD</sub>**

For instruction address range matches, an instruction at address A matches if the address A lies between the lower comparator address, which is specified by comparator <n-1>, and the limit comparator address, which is specified by comparator <n>. Both addresses are inclusive to the range.

*The extension requirements are - DWT-T.*

**R<sub>JNXZ</sub>**

For instruction address range matches, if the instruction at address A is a 4-byte T32 instruction, and the address A+2 lies in the range but the address A does not lie in the range, it is UNPREDICTABLE whether a match is generated.

*The extension requirements are - DWT-T.*

**R<sub>MLMQ</sub>**

For instruction address range matches, if so configured, a Data trace PC value packet or Data trace match packet is generated for the first instruction that is executed in the range.

*The extension requirements are - DWT-T.*

**I<sub>VHHW</sub>**

For instruction address range matches, if so configured, a branch or sequential execution that stays within the range does not necessarily generate a new packet.

*The extension requirements are - DWT-T.*

**R<sub>HMNX</sub>**

For instruction address range matches, if so configured, **CMPMATCH**[n-1] is triggered for each instruction that is executed inside the range, where n-1 is the lower of the two comparators that configure the range.

*The extension requirements are - DWT-T.*

## Data address matching in detail

- R<sub>BPWC</sub>** For all Data Address matches, all bits of the comparator address are considered.  
*The extension requirements are - DWT-T.*
- R<sub>GSLX</sub>** Speculative reads might generate data address matches.  
*The extension requirements are - DWT-T.*
- R<sub>WBBH</sub>** Speculative writes do not generate data address matches.  
*The extension requirements are - DWT-T.*
- R<sub>VJFB</sub>** Prefetches into a cache do not generate data address matches.  
*The extension requirements are - DWT-T.*
- R<sub>CMRP</sub>** For single data address matches, an access matches if any accessed byte lies between the comparator address and a limit that is defined by [DWT\\_FUNCTION.DATAVSIZE](#).  
*The extension requirements are - DWT-T.*
- R<sub>KHRF</sub>** For single data address matches, the comparator address is naturally aligned to [DWT\\_FUNCTION.DATAVSIZE](#) otherwise generation of watchpoint events is UNPREDICTABLE.  
*The extension requirements are - DWT-T.*
- R<sub>KKRJ</sub>** For data address range matches, an access matches if any accessed byte lies between the lower comparator address, which is specified by comparator <n-1>, and the limit comparator address, which is specified by comparator <n>. Both addresses are inclusive to the range.  
*The extension requirements are - DWT-T.*
- R<sub>CFMR</sub>** For data address range matches, [DWT\\_FUNCTION.DATAVSIZE](#) is set to 0b00 for both the lower comparator address and the limit comparator address otherwise it is UNPREDICTABLE whether or not a match is generated.  
*The extension requirements are - DWT-T.*
- Data value matching in detail**
- R<sub>BMSM</sub>** Data value matching is only possible if the Main Extension is implemented.  
*The extension requirements are - DWT-T.*
- R<sub>FVFO</sub>** Speculative reads might generate data value matches.  
*The extension requirements are - DWT-T.*
- R<sub>VGJF</sub>** Speculative writes do not generate data value matches.  
*The extension requirements are - DWT-T.*
- R<sub>MLFK</sub>** Prefetches into a cache do not generate data value matches.  
*The extension requirements are - DWT-T.*
- R<sub>RMDB</sub>** For data value matches, if the access size is smaller than [DWT\\_FUNCTION.DATAVSIZE](#), there is no match.  
*The extension requirements are - DWT-T.*
- R<sub>ZDPM</sub>** For unlinked data value matches, an access matches if all bytes of any naturally-aligned subset, the size of which is specified by [DWT\\_FUNCTION.DATAVSIZE](#), of the access match the data value in [DWT\\_COMPn](#).  
*The extension requirements are - DWT-T.*

- R<sub>ZHXP</sub>** The data value in [DWT\\_COMPn](#) is in little-endian order with respect to memory.  
*The extension requirements are - DWT-T.*
- I<sub>HMS</sub>** If the access is unaligned then this might generate a higher priority alignment fault, depending on the instruction type, profile, and configuration. In these cases no match is generated.  
*The extension requirements are - DWT-T.*
- R<sub>SQKS</sub>** For unlinked data value matches, if an access is unaligned, it is IMPLEMENTATION DEFINED whether it either treated as:
- A sequence of byte accesses.
  - A sequence of naturally-aligned accesses covering the accessed bytes. For a read, this access might access more bytes than the original access.
- The extension requirements are - DWT-T.*
- R<sub>QRPW</sub>** For linked data value matching, if an access is larger than [DWT\\_FUNCTION.DATAVSIZE](#), then only the naturally-aligned subset of the access of size [DWT\\_FUNCTION.DATAVSIZE](#) at the matching address is compared for a match.  
*The extension requirements are - DWT-T.*
- R<sub>QVRK</sub>** For linked data value matching, the data address comparator address is naturally aligned to [DWT\\_FUNCTION.DATAVSIZE](#), and the [DWT\\_FUNCTION.DATAVSIZE](#) values for both comparators are the same.  
*The extension requirements are - DWT-T.*
- R<sub>KRCV</sub>** A Data value comparator that is linked to a Data address comparator does not change the behavior of the address comparator.  
*The extension requirements are - DWT-T.*
- R<sub>KQJB</sub>** For each comparator *n* that is configured to Data Value or Linked Data Value matching it is UNPREDICTABLE whether comparator *n* generates a match when for bit *m*=31-0, if any of the following are true:
- [DWT\\_FUNCTION<n>.DATAVSIZE](#) specifies halfword or byte comparison and [DWT\\_COMPn\[31:16\]](#) is not equal to [DWT\\_COMPn\[15:0\]](#).
  - [DWT\\_FUNCTION<n>.DATAVSIZE](#) specifies byte comparison and [DWT\\_COMPn\[15:8\]](#) is not equal to [DWT\\_COMPn\[7:0\]](#).
- The extension requirements are - DWT-T.*

See also:

[DWT\\_AddressCompare\(\)](#).  
[DWT\\_ValidMatch\(\)](#).  
[DWT\\_InstructionAddressMatch\(\)](#).  
[DWT\\_DataAddressMatch\(\)](#).  
[DWT\\_DataValueMatch\(\)](#).

## B12.2.6 DWT match restrictions and relaxations

- R<sub>FRWG</sub>** It is IMPLEMENTATION DEFINED whether the DWT treats a fetch from the exception vector table as part of an exception entry or reset as a data access or ignores these accesses, for the purposes of DWT comparator matches.

*The extension requirements are - DWT-T.*

**R<sub>DTHW</sub>** A fetch by the DWT from the exception vector table as part of an exception entry is never treated as an instruction fetch.

*The extension requirements are - DWT-T.*

**R<sub>JQHW</sub>** If a return is tail-chained, it is IMPLEMENTATION DEFINED whether hardware accesses the stack and therefore IMPLEMENTATION DEFINED whether the DWT can generate events or trace.

*The extension requirements are - DWT-T.*

**R<sub>VJTK</sub>** The DWT does not match accesses from the DAP.

*The extension requirements are - DWT-T.*

**R<sub>MNBX</sub>** Any executed NOP or IT that matches an appropriately configured instruction address watchpoint causes a match.

*The extension requirements are - DWT-T.*

**R<sub>SLPX</sub>** It is IMPLEMENTATION DEFINED whether a failed STREX instruction can generate a data access match.

*The extension requirements are - DWT-T.*

**R<sub>NHLN</sub>** If an instruction or operation makes multiple or unaligned data accesses, then it is UNPREDICTABLE whether any nonmatching access generated by an instruction that generated a matching access is treated as a matching access.

*The extension requirements are - DWT-T.*

**R<sub>CSSQ</sub>** If an instruction or operation makes multiple or unaligned data accesses, then CMPMATCH is triggered for each matching access.

*The extension requirements are - DWT-T.*

**R<sub>VFXT</sub>** If an instruction or operation makes multiple or unaligned data accesses, then, if so configured, only a data value match of at least a part of the value that is guaranteed to be single-copy atomic can generate a match.

*The extension requirements are - DWT-T.*

**R<sub>WJNR</sub>** If an instruction or operation makes multiple or unaligned data accesses, then, if so configured, for a matching data access that generates a debug event, if permitted, DHCSR.C\_HALT or DEMCR.MON\_PEND, as applicable, is set to 1.

A pending DebugMonitor exception does not interrupt the multiple accesses, but another interrupt might, which means that the debug event might be taken before the multiple operations complete.

*The extension requirements are - DWT-T.*

**R<sub>QCJL</sub>** The DWT can match on the address of an access that generates a BusFault.

*The extension requirements are - DWT-T.*

**R<sub>QVHL</sub>** It is IMPLEMENTATION DEFINED whether a stored value for an access that generates a BusFault:

- Can generate a data value match.
- Can be traced.

*The extension requirements are - DWT-T.*

**R<sub>KLFC</sub>** For a load access that returns a BusFault, any data that is returned by the memory system is invalid, and the DWT does not:

- Generate a data value match.

- Generate a Data trace data value packet.

*The extension requirements are - DWT-T.*

**R<sub>TQCF</sub>** A data access that generates any fault other than a BusFault does not generate a data address or data value match at the DWT and is not traced.

*The extension requirements are - DWT-T.*

**R<sub>FRHP</sub>** DWT matches are generated asynchronously.

*The extension requirements are - DWT-T.*

**R<sub>THHR</sub>** A **DSB** barrier guarantees that the effect of a DWT match is visible to a subsequent read of **DWT\_FUNCTION-MATCHED**, **DHCSR**, or **DEMCR**. In the absence of a **DSB** barrier, the effect is only guaranteed to be visible in finite time.

*The extension requirements are - DWT-T.*

**R<sub>HPGH</sub>** The effects of a DWT match never affect instructions appearing in program order before the operation that generates the match.

*The extension requirements are - DWT-T.*

See also:

[B3.26 Tail-chaining on page 108.](#)

## B12.2.7 DWT trace restrictions and relaxations

**R<sub>PGCS</sub>** If a single instruction makes multiple single-copy atomic accesses, such as the multiple-byte accesses from an unaligned access or a predicated vector load/store operation, the DWT might gather a sequence of consecutive bytes from the multiple accesses, and trace it as a single access.

*The extension requirements are - DWT-T.*

**R<sub>HDKK</sub>** Where a single instruction or operation, or multiple instructions, generate multiple accesses that each generate one or more trace packets, then if the architecture guarantees the order in which a pair of these accesses is observed by the PE, the first trace packets that are generated for each of those accesses appear in the trace output in the same order.

*The extension requirements are - DWT-T.*

**R<sub>WSKK</sub>** Where a single instruction or operation, or multiple instructions, generate multiple accesses that each generate one or more trace packets, then if the architecture does not guarantee the order of the accesses, the order of the trace packets in the trace output is not defined.

*The extension requirements are - DWT-T.*

**R<sub>XCNB</sub>** If a single instruction or operation makes multiple or unaligned data accesses, then, if so configured, only the first access is guaranteed to generate a Data trace PC value packet, Data trace data address packet, or Data trace match packet. If the architecture does not guarantee the order of the accesses, the first access might be any of the accesses.

*The extension requirements are - DWT-T.*

**R<sub>XVBT</sub>** If a single instruction or operation makes multiple or unaligned data accesses, then, if so configured, a Data trace data value packet is generated for each matching access.

*The extension requirements are - DWT-T.*

**R<sub>QSCF</sub>** If a single instruction or operation makes unaligned data accesses, it is UNPREDICTABLE how many Data trace data value packets are generated for each unaligned matching access. An implementation might overread, meaning that more data outside the access might be traced.

*The extension requirements are - DWT-T.*

**R<sub>KXBL</sub>** If a single instruction or operation makes multiple or unaligned data accesses, then, if so configured, for a matching data access that generates a Data trace data value packet, at least that part of the value that is guaranteed to be single-copy atomic is traced.

*The extension requirements are - DWT-T.*

**R<sub>QWQS</sub>** Duplicate Data trace PC value packets, Data trace data address packets, and Data trace data value packets from a single access are not generated for a single access.

*The extension requirements are - DWT-T.*

**R<sub>CPXW</sub>** Where a comparator or linked pair of comparators generates multiple packet types for a single access, the packets appear in the trace output in the following order:

1. Data trace PC value packet.
2. Data trace match packet, generated by a Data address or Data address with value comparator match.
3. Data trace data address packet.
4. Data trace match packet, generated by a Data value comparator match.
5. Data trace data value packet.

*The extension requirements are - DWT-T.*

**R<sub>QXBC</sub>** Where a comparator or linked pair of comparators generates multiple packet types for a single access, packets are not interleaved with packets that are generated by other accesses by the same comparator or linked pair of comparators.

*The extension requirements are - DWT-T.*

**R<sub>RHNF</sub>** Where a comparator or linked pair of comparators generates a trace packet for a single access, if a comparator other than this comparator or this linked pair of comparators generates a trace packet of the same type for the same access, then only one of these packets is output. It is IMPLEMENTATION DEFINED which comparator is chosen.

*The extension requirements are - DWT-T.*

**I<sub>MJXG</sub>** Arm recommends that the packet from the lowest-numbered comparator is output.

*The extension requirements are - DWT-T.*

**R<sub>DKMV</sub>** Where a comparator or linked pair of comparators generates multiple packet types for a single access, if any of the packets cannot be output and an Overflow packet is generated, then the remaining packets for that access are not generated.

*The extension requirements are - DWT-T.*

**R<sub>LNBW</sub>** Where a comparator or linked pair of comparators generates multiple packet types for a single access, packets might be interleaved with packets that are generated for the same access by comparators other than this comparator or this linked pair of comparators.

*The extension requirements are - DWT-T.*

## B12.2.8 CYCCNT cycle counter and related timers

**R<sub>SVPW</sub>** CYCCNT is an optional free-running 32-bit cycle counter. If the DWT unit implements CYCCNT then [DWT\\_CTRL.NOCYCCNT](#) is RAZ.



The extension requirements are - *DWT-T*.

**R<sub>KRFP</sub>** When implemented and enabled, CYCCNT increments on each cycle of the PE clock.

The extension requirements are - *DWT-T*.

**R<sub>NFJW</sub>** When the counter overflows it transparently wraps to zero.

The extension requirements are - *DWT-T*.

**R<sub>GXJK</sub>** [DWT\\_CTRL.CYCCNTENA](#) enables the CYCCNT counter.

The extension requirements are - *DWT-T*.

**R<sub>BKCG</sub>** POSTCNT is a 4-bit countdown counter derived from CYCCNT, that acts as a timer for the periodic generation of Periodic PC sample packets or Event counter packets, when these packets are enabled.

The extension requirements are - *DWT-T*.

**I<sub>MGGL</sub>** Periodic PC sample packets are not the same as the Data trace PC value packets that are generated by the DWT comparators.

The extension requirements are - *DWT-T*.

**R<sub>DKGR</sub>** The DWT does not support the generation of Periodic PC sample packets or Event packets if it does not implement the CYCCNT timer and [DWT\\_CTRL.NOTRCPKT](#) is RAO.

The extension requirements are - *DWT-T*.

**R<sub>RNTV</sub>** The [DWT\\_CTRL.CYCTAP](#) bit selects the CYCCNT tap bit for POSTCNT.

| CYCTAP bit | CYCCNT tap at | POSTCNT clock rate |
|------------|---------------|--------------------|
| 0          | Bit[6]        | (PE clock)/64      |
| 1          | Bit[10]       | (PE clock)/1024    |

The extension requirements are - *DWT-T*.

**R<sub>SXKK</sub>** A write to [DWT\\_CTRL](#) will initialize POSTCNT to the previous value of [DWT\\_CTRL.POSTINIT](#) if all of the following are true:

- [DWT\\_CTRL.PCSAMPLENA](#) was set to 0 prior to the write.
- [DWT\\_CTRL.CYCEVTENA](#) was set to 0 prior to the write.
- The write sets either [DWT\\_CTRL.PCSAMPLENA](#) or [DWT\\_CTRL.CYCEVTENA](#) to 1.

It is UNPREDICTABLE whether any other write to [DWT\\_CTRL](#) that alters the value of [DWT\\_CTRL.PCSAMPLENA](#) and [DWT\\_CTRL.CYCEVTENA](#) sets POSTCNT to [DWT\\_CTRL.POSTINIT](#) or leaves POSTCNT unchanged.

The extension requirements are - *DWT-T*.

**R<sub>XFRM</sub>** When either [DWT\\_CTRL.PCSAMPLENA](#) or [DWT\\_CTRL.CYCEVTENA](#) is set to 1, and the CYCCNT tap bit transitions, either from 0 to 1 or from 1 to 0:

- If POSTCNT is nonzero, POSTCNT decrements by 1.
- If POSTCNT is 0, the DWT:
  - Reloads POSTCNT from [DWT\\_CTRL.POSTPRESET](#).
  - Generates a Periodic PC Sample packets if [DWT\\_CTRL.PCSAMPLENA](#) is set to 1.
  - Generates an Event Counter packet with the Cyc bit set to 1 if [DWT\\_CTRL.CYCEVTENA](#) is set to 1.

The extension requirements are - *DWT-T*.

**I<sub>PNN</sub>** The enable bit for the POSTCNT counter underflow event is [DWT\\_CTRL.CYCEVTENA](#). There is no overflow event for the CYCCNT counter. When CYCCNT overflows it wraps to zero transparently. Software cannot access the POSTCNT value directly, or change this value.

*The extension requirements are - DWT-T.*

**I<sub>JRV</sub>** This means that, to initialize POSTCNT, software:

1. Ensures that [DWT\\_CTRL.CYCEVTENA](#) and [DWT\\_CTRL.PCSAMPLENA](#) are set to 0. This can be achieved with a single write to [DWT\\_CTRL](#). This is also the reset value of these bits.
2. Writes the required initial value of POSTCNT to the [DWT\\_CTRL.POSTINIT](#) field, leaving [DWT\\_CTRL.CYCEVTENA](#) and [DWT\\_CTRL.PCSAMPLENA](#) set to 0.
3. Sets either [DWT\\_CTRL.CYCEVTENA](#) or [DWT\\_CTRL.PCSAMPLENA](#) to 1 to enable the POSTCNT counter.

Each of these are separate writes to [DWT\\_CTRL](#).

*The extension requirements are - DWT-T.*

**R<sub>KNH</sub>** Disabling CYCCNT stops POSTCNT.

*The extension requirements are - DWT-T.*

**R<sub>TMH</sub>** Writes to [DWT\\_CTRL.POSTINIT](#) are ignored if either [DWT\\_CTRL.CYCEVTENA](#) was set to 1 or [DWT\\_CTRL.PCSAMPLENA](#) was set to 1 prior to the write.

*The extension requirements are - DWT-T.*

## B12.2.9 Profiling counter support

**I<sub>HXP</sub>** If the Main Extension is implemented profiling counter support is an optional Non-invasive debug feature.

*The extension requirements are - DWT-T && M.*

**R<sub>WHW</sub>** If profiling counter support is implemented the DWT provides five 8-bit Event counters for software profiling:

- [DWT\\_FOLDCNT](#).
- [DWT\\_LSUNCT](#).
- [DWT\\_EXCCNT](#).
- [DWT\\_SLEPCNT](#).
- [DWT\\_CPICNT](#).

*The extension requirements are - DWT-T.*

**R<sub>GLM</sub>** Event counters do not increment when the PE is halted.

*The extension requirements are - DWT-T.*

**R<sub>BRG</sub>** The Event counters provide broadly accurate and statistically useful count information. However, the architecture allows for a reasonable degree of inaccuracy in the counts.

*The extension requirements are - DWT-T.*

**R<sub>WMN</sub>** The Event counters use the same definition of cycle in particular when counting cycles in power-saving modes.

*The extension requirements are - DWT-T && M.*

**I<sub>GNW</sub>** To keep the implementation and validation cost low, a reasonable degree of inaccuracy in the counts is acceptable. Arm does not define *a reasonable degree of inaccuracy* but recommends the following guidelines:

- Under normal operating conditions, the Event counters present an accurate value count.

- Entry to or exit from Debug state can be a source of inaccuracy.
- Under very unusual, non-repeating pathological cases, the counts can be inaccurate.

An implementation does not introduce inaccuracies that can be triggered systematically by the execution of normal pieces of software. As the Event counters include counters for measuring exception overhead, this includes the operation of exceptions.

*The extension requirements are - DWT-T.*

**I<sub>CHKR</sub>** Arm strongly recommends that an implementation document any particular scenarios where significant inaccuracies in the Event counters are expected.

*The extension requirements are - DWT-T.*

**I<sub>MWGO</sub>** At entry and exit from an exception or sleep state, the exact attribution of cycles to the exception and cycles to the sleep overhead counters is IMPLEMENTATION DEFINED. Arm recommends that the overhead cycles are attributed to the overhead counters.

*The extension requirements are - DWT-T.*

**I<sub>MPQN</sub>** The architecture does not define the point in a pipeline where the particular instruction increments an Event counter, relative to the point where the incremented counter can be read.

*The extension requirements are - DWT-T.*

**R<sub>LMPK</sub>** An Event counter overflows on every 256th event that is counted and then wraps to 0. If the appropriate counter overflow event is enabled in **DWT\_CTRL** the DWT outputs an Event counter packet with the appropriate counter flag set to 1.

*The extension requirements are - DWT-T.*

**R<sub>LHMB</sub>** Setting one of the enable bits to 1 clears the corresponding counter to 0.

*The extension requirements are - DWT-T.*

**I<sub>QRPQ</sub>** The following equation holds:

$$ICNT = CNT_{CYCLES} + CNT_{FOLD} - (CNT_{LSU} + CNT_{EXC} + CNT_{SLEEP} + CNT_{CPI})$$

Where:

**ICNT**: is the total number of instructions [Architecturally executed](#).

**CNT<sub>CYCLES</sub>**: is the number of cycles counted by **DWT\_CYCCNT**.

**CNT<sub>FOLD</sub>**: is the number of instructions counted by **DWT\_FOLDCNT**.

**CNT<sub>LSU</sub>**: is the number of cycles counted by **DWT\_LSUNCT**.

**CNT<sub>EXC</sub>**: is the number of cycles counted by **DWT\_EXCCNT**.

**CNT<sub>SLEEP</sub>**: is the number of cycles counted by **DWT\_SLEEPCNT**.

**CNT<sub>CPI</sub>**: is the number of cycles counted by **DWT\_CPICNT**.

*The extension requirements are - DWT-T.*

See also:

[B12.4 Trace Port Interface Unit on page 299](#).

## Generating Overflow packets from Event counters

**R<sub>KWDH</sub>** If an Event counter wraps to zero and the previous Event counter packet has been delayed and has not yet been output, and the counter flag in the previous Event counter packet is set to 0, then it is IMPLEMENTATION DEFINED whether:

- The DWT attempts to generate a second Event counter packet.
- The DWT updates the delayed Event counter packet to include the new wrap event.

*The extension requirements are - DWT-T.*

**R<sub>HKTL</sub>** If an Event counter wraps to zero and the previous Event counter packet has been delayed and has not yet been output, and the counter flag in the previous Event counter packet is set to 1, the DWT attempts to generate a second Event counter packet.

*The extension requirements are - DWT-T.*

**R<sub>VPXK</sub>** If the DWT unit attempts to generate a packet when its output buffer is full, an Overflow packet is output.

*The extension requirements are - DWT-T.*

**R<sub>SFFL</sub>** The size of the DWT output buffer is IMPLEMENTATION DEFINED.

*The extension requirements are - DWT-T.*

### B12.2.10 Program Counter sampling support

**R<sub>FXWL</sub>** Program Counter sampling is an optional component provided through [DWT\\_PCSR](#).

*The extension requirements are - DWT-T.*

**I<sub>LNJL</sub>** Program Counter sampling is independent of PC sampling provided by:

- Periodic PC sample packets.
- Data trace PC value packets generated as a result of a DWT comparator match.

*The extension requirements are - DWT-T.*

**I<sub>KVFB</sub>** The architecture does not define the delay between an instruction being executed by the PE and its address being written to [DWT\\_PCSR](#).

*The extension requirements are - DWT-T.*

**R<sub>NGNT</sub>** When [DWT\\_PCSR](#) returns a value other than 0xFFFFFFFF, the returned value is an instruction that has been committed for execution. It is IMPLEMENTATION DEFINED whether an instruction that failed its condition code check is considered as committed for execution. A read of [DWT\\_PCSR](#) does not return the address of an instruction that has been fetched but not committed for execution.

*The extension requirements are - DWT-T.*

**I<sub>KCBH</sub>** Arm recommends that instructions that fail the condition code check are considered as committed instructions.

*The extension requirements are - DWT-T.*

**R<sub>WPMF</sub>** [DWT\\_PCSR](#) is able to sample references to branch targets. It is IMPLEMENTATION DEFINED whether it can sample references to other instructions.

*The extension requirements are - DWT-T.*

**I<sub>SJVK</sub>** Arm recommends that [DWT\\_PCSR](#) can sample a reference to any instruction.

*The extension requirements are - DWT-T.*

- R<sub>LMDG</sub>** The branch target for a conditional branch that fails its Condition code check is the instruction that immediately follows the conditional branch instruction. The branch target for an exception is the exception vector address.  
*The extension requirements are - DWT-T.*
- R<sub>NWKP</sub>** Periodic sampling of [DWT\\_PCSR](#) provides broadly accurate and statistically useful profile information. However, the architecture allows for a reasonable degree of inaccuracy in the sampled data.  
*The extension requirements are - DWT-T.*
- I<sub>TJTS</sub>** To keep the implementation and validation cost low, a reasonable degree of inaccuracy in the counts is acceptable. Arm does not define *a reasonable degree of inaccuracy* but recommends the following guidelines:
- In exceptional circumstances, such as a change in Security state or other boundary condition, it is acceptable for the sample to represent an instruction that was not committed for execution.
  - Under unusual non-repeating pathological cases, the sample can represent an instruction that was not committed for execution. These cases are likely to occur as a result of asynchronous exceptions, such as interrupts, where the chance of a systematic error in sampling is very unlikely.
- The extension requirements are - DWT-T.*
- I<sub>KVJM</sub>** Arm strongly recommends that an implementation document any particular scenarios where significant inaccuracies in the sampled data are expected.  
*The extension requirements are - DWT-T.*
- R<sub>JMVS</sub>** When [DEMCR.TRCENA](#) is set to 0 any read of [DWT\\_PCSR](#) returns an UNKNOWN value.  
*The extension requirements are - DWT-T.*
- I<sub>PXMR</sub>** A read of [DWT\\_PCSR](#) will return 0xFFFFFFFF if any of the following are true:
- The PE is in Debug state.
  - The instruction was executed in Secure state and [SecureNoninvasiveDebugAllowed\(\)](#) returns FALSE.
  - [NoninvasiveDebugAllowed\(\)](#) returns FALSE.
  - The address of a recently executed instruction is not available.
- The extension requirements are - DWT-T. Note, S is required for Secure state.*

## B12.3 Embedded Trace Macrocell

**I<sub>LCCX</sub>** An *Embedded Trace Macrocell* (ETM) is an optional non-invasive debug feature of an Armv8-M implementation.  
 The extension requirements are - *ETM*.

**R<sub>NGTT</sub>** An ETM implementation complies with one of the following versions of the ETM architecture:

|                 | Data trace | Security Extension          |                             |
|-----------------|------------|-----------------------------|-----------------------------|
|                 |            | Implemented                 | Not implemented             |
| Implemented     |            | ETMv3 not permitted         | ETMv3 not permitted         |
|                 |            | ETMv4, version 4.2 or later | ETMv4, version 4.0 or later |
| Not Implemented |            | ETMv3, version 3.5 or later | ETMv3, version 3.5          |
|                 |            | ETMv4, version 4.2 or later | ETMv4, version 4.0 or later |

The extension requirements are - *ETM*.

**R<sub>LPJM</sub>** If an ETM is implemented a trace sink is also implemented. If the trace sink that is implemented is the TPIU it is CoreSight compliant, and complies with the TPIU architecture for compatibility with Arm and other CoreSight-compatible debug solutions.

The extension requirements are - *ETM*.

**R<sub>NLNS</sub>** When an Armv8-M implementation includes an ETM, the **CMPMATCH[N]** signals from the DWT unit are available as control inputs to the ETM unit.

The extension requirements are - *ETM*.

**R<sub>NJDK</sub>** If the Main Extension is not implemented, it is IMPLEMENTATION DEFINED whether the ETM is accessible only to the debugger and is RES0 to software.

The extension requirements are - *ETM* && *!M*.

**R<sub>WPBN</sub>** If the ETMv3 is implemented the debugger programs the ETMTRACEIDR with a unique nonzero Trace ID for the ETM trace stream.

The extension requirements are - *ETM*.

**R<sub>TJSF</sub>** If the ETMv4 is implemented the debugger programs the TRCTRACEIDR with a unique nonzero Trace ID for the ETM trace stream.

The extension requirements are - *ETM*.

**R<sub>WSTB</sub>** The ETM is not directly affected by **DEMCR.TRCENA** being set to 0.

The extension requirements are - *ETM*.

See also:

*Arm® CoreSight™ Architecture Specification.*

[B12.2.4 CMPMATCH trigger events on page 286.](#)

## B12.4 Trace Port Interface Unit

- I<sub>PWXP</sub>** The *Trace Port Interface Unit* (TPIU) support for Armv8-M provides an output path for trace data from the DWT, ITM, and ETM. The TPIU is a trace sink.  
*The extension requirements are - TPIU.*
- R<sub>CRTQ</sub>** It is IMPLEMENTATION DEFINED whether the TPIU supports a parallel trace port output.  
*The extension requirements are - TPIU.*
- R<sub>GTRP</sub>** It is IMPLEMENTATION DEFINED whether the TPIU supports low-speed asynchronous serial port output using NRZ encoding. This operates as a traditional UART.  
*The extension requirements are - TPIU.*
- R<sub>LKQT</sub>** It is IMPLEMENTATION DEFINED whether the TPIU supports medium-speed asynchronous serial port output using Manchester encoding.  
*The extension requirements are - TPIU.*
- I<sub>SDDK</sub>** Arm recommends that the TPIU provides both parallel and asynchronous serial ports, for maximum flexibility with external capture devices.  
*The extension requirements are - TPIU.*
- R<sub>HJXK</sub>** Whether the trace port clock is synchronous to the PE clock is IMPLEMENTATION DEFINED.  
*The extension requirements are - TPIU.*
- R<sub>PKKS</sub>** It is IMPLEMENTATION DEFINED whether the TPIU is reset by a Cold reset or has an independent Cold reset.  
*The extension requirements are - TPIU.*
- R<sub>JBKJ</sub>** Software ensures that all trace is output and flushed to the trace sink before setting the [DEMCR.TRCENA](#) bit to 0.  
*The extension requirements are - TPIU.*
- R<sub>STLV</sub>** The TPIU is not directly affected by [DEMCR.TRCENA](#) being set to 0 or `NoninvasiveDebugAllowed()` being FALSE.  
*The extension requirements are - TPIU.*
- R<sub>JLCQ</sub>** The output formatting modes that are supported by the TPIU are IMPLEMENTATION DEFINED. They are:
- Bypass.
  - Continuous.
- The extension requirements are - TPIU.*
- R<sub>DMFP</sub>** Bypass mode is only supported if a serial port output is supported.  
*The extension requirements are - TPIU.*
- R<sub>RRJP</sub>** Continuous mode is supported if the parallel trace port is implemented. Continuous mode is selected when the parallel trace port is used.  
*The extension requirements are - TPIU.*
- R<sub>FCFT</sub>** Continuous mode is supported if the ETM is implemented. Continuous mode is selected when the ETM is used.  
*The extension requirements are - TPIU.*

See also:

*TPIU\_FFCR, Formatter and Flush Control Register.*

*B12.1 Instrumentation Trace Macrocell on page 269.*

*B12.3 Embedded Trace Macrocell on page 298.*

*Chapter B1 Resets on page 46.*



## B12.5 Flash Patch and Breakpoint unit

### B12.5.1 About the FPB unit

- R<sub>FTWL</sub>** The *Flash Patch and Breakpoint (FPB)* unit supports setting breakpoints on instruction fetches.  
*The extension requirements are - FPB.*
- I<sub>BPFS</sub>** The name Flash Patch and Breakpoint unit is historical and the architecture does not support remapping functionality.  
*The extension requirements are - FPB.*
- R<sub>GDWW</sub>** The number of implemented instruction address comparators is IMPLEMENTATION DEFINED. Software can discover the number of implemented instruction address comparators from [FP\\_CTRL.NUM\\_CODE](#).  
*The extension requirements are - FPB.*

See also:

[Chapter B6 The System Address Map](#) on page 198.

[B12.2.7 DWT trace restrictions and relaxations](#) on page 291.

[Chapter D1 Register Specification](#) on page 841.

### B12.5.2 FPB unit operation

- R<sub>RKFD</sub>** The FPB contains the following register types:
- A general control register, [FP\\_CTRL](#).
  - Comparator registers.
- The extension requirements are - FPB.*
- R<sub>BKKW</sub>** Each implemented instruction address comparator supports breakpoint generation.  
*The extension requirements are - FPB.*
- R<sub>FNQF</sub>** The [FP\\_CTRL](#) register provides a global enable bit for the FPB, and ID fields that indicate the numbers of instruction address comparison and literal comparison registers implemented.  
*The extension requirements are - FPB.*
- R<sub>CKBL</sub>** When configured for breakpoint generation, instruction address comparators can be configured to match any halfword-aligned addresses in the whole system address map.  
*The extension requirements are - FPB.*
- R<sub>XPXS</sub>** Instruction address comparators match only on instruction fetches. The FPB treats hardware accesses to the stack as data accesses for registers that are:
- Pushed to the stack by hardware as part of an exception entry or lazy state preservation.
  - Popped from the stack by hardware as part of an exception return.
  - Pushed to the stack by hardware as part of a Non-secure function return.
  - Popped from the stack by hardware as part of a Non-secure function call.

It is IMPLEMENTATION DEFINED whether the FPB treats a fetch from the exception vector table as part of an exception entry as a data access, or ignores these accesses, for the purposes of FPB address comparator matches. The fetch is never be treated as an instruction fetch.

The FPB does not match accesses from the DAP.

*The extension requirements are - FPB.*

**I<sub>CNBW</sub>** Bit[0] of each instruction fetch address is always 0.

*The extension requirements are - FPB.*

**R<sub>CJJK</sub>** When an Instruction address matching comparator is configured for breakpoint generation, a match on the address of a 32-bit instruction is configured to match the first halfword or both halfwords of the instruction.

*The extension requirements are - FPB.*

**R<sub>WSXN</sub>** If a Breakpoint debug event is generated by the FPB on the second halfword of a 32-bit T32 instruction, it is UNPREDICTABLE whether the breakpoint generates a debug event.

*The extension requirements are - FPB.*

**R<sub>XKJW</sub>** An FPB match specifying a Breakpoint debug event generates a Breakpoint debug event that halts the PE if all of the following conditions are true:

- `HaltingDebugAllowed() == TRUE`.
- `DHCSR.C_DEBUGEN == 1`.
- `DHCSR.S_HALT == 0`.
- The Security Extension is not implemented, the matching instruction is executed in Non-secure state, or `DHCSR.S_SDE == 1`.

*The extension requirements are - FPB.*

**R<sub>HXMP</sub>** An FPB match specifying a Breakpoint debug event generates a DebugMonitor exception if it does not halt the PE and all of the following conditions are true:

- `DEMCR.MON_EN == 1`.
- `DHCSR.S_HALT == 0`.
- The DebugMonitor exception group priority is greater than the current execution priority.
- The Security Extension is not implemented, the matching instruction is executed in Non-secure state, or `DEMCR.SDME == 1`.

*The extension requirements are - FPB.*

**R<sub>BFPK</sub>** An FPB match that specifies a Breakpoint debug event is ignored if it does not meet the conditions for generating either:

- A Breakpoint debug event that halts the PE.
- A DebugMonitor exception.

*The extension requirements are - FPB.*

**R<sub>CLNV</sub>** Between a change to the debug authentication interface, `DHCSR` or `DEMCR`, that disables debug, and a following context synchronization event, it is UNPREDICTABLE whether any breakpoints generated by the FPB:

- Generate a Breakpoint debug event based on the old values and either:
  - If the Main Extension is implemented, generate a DebugMonitor exception.
  - Halts the PE.
- Are ignored.

*The extension requirements are - FPB.*

See also:

[Halting debug.](#)

[B11.4.1 About debug events on page 250.](#)

`BKPTInstrDebugEvent()`

`FPB_BreakpointMatch()`

### B12.5.3 Cache maintenance

$R_{BWSW}$  Instruction caches are not permitted to cache breakpoints that are generated by a Flash Patch and Breakpoint unit.

*The extension requirements are - FPB.*

**Part C**  
**Armv8-M Instruction Set**

## Chapter C1

# Instruction Set Overview

This chapter provides a definition of the *instruction descriptions* contained in [Chapter C2 Instruction Specification](#) on page 327. It contains the following sections:

- [C1.1 Instruction set](#) on page 306.
- [C1.2 Format of instruction descriptions](#) on page 307.
- [C1.3 Conditional execution](#) on page 313.
- [C1.4 Instruction set encoding information](#) on page 319.
- [C1.5 Modified immediate constants](#) on page 324.
- [C1.6 NOP-compatible hint instructions](#) on page 325.
- [C1.7 SBZ or SBO fields in instructions](#) on page 326.

## C1.1 Instruction set

$R_{NPFK}$  There is one instruction set, called T32.

See also:

[C1.4 Instruction set encoding information on page 319.](#)

[Chapter C2 Instruction Specification on page 327.](#)

## C1.2 Format of instruction descriptions

**I<sub>XQQV</sub>** Each instruction description in [Chapter C2 Instruction Specification on page 327](#) has the following content:

1. A title.
2. A short description.
3. The instruction encoding or encodings.
4. Any alias conditions, if applicable.
5. A list of the assembler symbols for the instruction.
6. Pseudocode describing how the instruction operates.
7. Notes, if applicable.

### C1.2.1 The title

**I<sub>RFFL</sub>** The title of an instruction description includes the base mnemonic or mnemonics for the instruction. This is part of the assembler syntax, for example SUB.

**I<sub>BSWN</sub>** If different forms of an instruction use the same base mnemonic, each form has its own description. In this case, the title is the mnemonic followed by a short description of the instruction form in parentheses. This is most often used when an operand is an immediate value in one instruction form, but is a register in another form.

For example, in [Chapter C2 Instruction Specification on page 327](#) the Armv8-M Instruction Set there are the following titles for different forms of the ADD instruction:

- ADD (SP plus immediate)
- ADD (SP plus register)
- ADD (immediate)
- ADD (immediate to PC)
- ADD (register)

**I<sub>RKXC</sub>** Where an instruction has more than one variant, the descriptions might be combined, for example for CDP and CDP2.

### C1.2.2 A short description

**I<sub>QNXW</sub>** This briefly describes the function of the instruction. The short description is not a complete description of the instruction and must be read in conjunction with the instruction encoding, mnemonic, alias conditions, assembler symbols, pseudocode and any applicable notes.

### C1.2.3 The instruction encoding or encodings

**R<sub>LTJB</sub>** Instruction descriptions in this manual contain:

- An encoding section, containing one or more encoding diagrams, each followed by some decode pseudocode that:
  1. Picks out any encoding-specific special cases.
  2. Translates the fields of the encoding into inputs for the common pseudocode of the instruction

- An operation section, containing common pseudocode that applies to all of the encodings being described. The Operation section pseudocode contains a call to the `EncodingSpecificOperations()` function which triggers the decode pseudocode, either at its start or only after a [Condition code check](#) performed by `if ConditionPassed() then`.

- R<sub>BDDV</sub>** An encoding diagram specifies each bit of the instruction encoding as one of the following:
- A mandatory 0 or 1, represented in the diagram as 0 or 1. If the PE attempts to decode and execute the instruction and a bit does not have a mandatory value, the encoding corresponds to a different instruction.
  - A *should be* 0 or *should be* 1, represented in the diagram as (0) or (1). If the PE attempts to decode and execute the instruction and a bit does not have the *should be* value, the instruction is CONSTRAINED UNPREDICTABLE.
  - A named single bit or a bit in a named multi-bit field.
- R<sub>BBZT</sub>** An encoding diagram matches an instruction if all mandatory bits are identical in the encoding diagram and the instruction.
- I<sub>DKWV</sub>** Between each encoding diagram and its T <n> heading, there is an italicized statement that describes which *Armv8-M variant* the encoding is present in. For example, *Armv8-M Main Extension only*.
- I<sub>JSBT</sub>** The instruction description shows the instruction encoding diagram, or, if the instruction has multiple encodings, shows all of the encoding diagrams. The heading for each encoding is the letter *T* followed by an arbitrary number, usually between 1 and 5.
- I<sub>FQDP</sub>** Below each encoding diagram is the *assembler syntax prototype* for that encoding, written in typewriter font. The assembler syntax prototype describes the syntax that can be used in the assembler to select this encoding, and also the syntax that is used when disassembling this encoding.
- I<sub>BLJR</sub>** In some cases an encoding has multiple variants of *assembler syntax prototype*, when the prototype differs depending on the value in one or more of the encoding fields. In these cases, the correct variant to use can be identified by either:
- Its subheading.
  - An annotation to the syntax.

See also:

[B5.3 Endianness on page 146.](#)

[C1.2.6 Pseudocode describing how the instruction operates on page 310.](#)

## C1.2.4 Any alias conditions, if applicable

- I<sub>BMHC</sub>** Alias conditions are an optional part of an instruction description. If included, it describes the set of conditions for which an alternative mnemonic and its associated assembler syntax prototypes are preferred for disassembly by a disassembler. It includes a link to the alias instruction description that defines the alternative syntax. The alias



syntax and the original syntax can be used interchangeably in the assembler source code.

**I<sub>RBCM</sub>** Arm recommends that if a disassembler outputs the alias syntax, it consistently outputs the alias syntax.

**I<sub>ZJKQ</sub>** Arm recommends that where possible, the alias is used.

### C1.2.5 Standard assembler syntax fields

**I<sub>RHCC</sub>** This manual uses the Arm *Unified Assembler Language* (UAL). This assembly language syntax provides a canonical form for all T32 instructions.

**I<sub>LBNB</sub>** **UAL** describes the syntax for the mnemonic and the operands of each instruction. Operands can also be referred to as *Assembler symbols*. In addition, **UAL** assumes that instructions and data items can be given labels. It does not specify the syntax to be used for labels, see the assembler documentation for these details.

**I<sub>DPLM</sub>** The *Assembler symbols* subsection of an instruction description contains a list of the symbols that the assembler syntax prototype or prototypes use.

The following conventions are used:

< >: Angle brackets. Any symbol enclosed by these is mandatory. For each symbol, there is a description of what the symbol represents. The description usually also specifies which encoding field or fields encodes the symbol.

{ }: Brace brackets. Any symbol enclosed by these is optional. For each optional symbol, there is a description of what the symbol represents and how its presence or absence is encoded.

In some assembler syntax prototypes, some brace brackets are mandatory, for example if they surround a register list. When the use of brace brackets is mandatory, they are separated from other syntax items by one or more spaces.

# : Usually precedes a numeric constant. All uses of # are optional in assembler source code. Arm recommends that disassemblers output the # where the assembler syntax prototype includes it.

+/-: Indicates an optional + or - sign. If neither is coded, + is assumed.

! : Indicates that the result address is written back to the base register.

**R<sub>MBQS</sub>** Single spaces are used for clarity, to separate syntax items. Where a space is mandatory, the assembler syntax prototype shows two or more consecutive spaces.

**R<sub>SXWN</sub>** Any characters not shown in this conventions list must be coded exactly as shown in the assembler syntax prototype. Apart from brace brackets, these characters are used as part of a meta-language to define the architectural assembler syntax prototype for an instruction encoding, but have no architecturally defined significance in the input to an assembler or in the output from a disassembler.

**R<sub>QZDB</sub>** **UAL** includes *instruction selection* rules that specify which instruction encoding is selected when more than one can provide the required functionality. The following assembler syntax prototype fields are standard across all or

most instructions:

<c>: Specifies the condition under which the instruction is executed. If <c> is omitted, it defaults to *always* (AL).

<q>: Specifies one of the following optional assembler qualifiers on the instruction:

.N

Meaning narrow. The assembler must select a 16-bit encoding for the instruction. If this is not possible, an assembler error is produced.

.W

Meaning wide. The assembler must select a 32-bit encoding for the instruction. If this is not possible, an assembler error is produced.

If neither .W nor .N is specified, the assembler can select either a 16-bit or 32-bit encoding. If both encoding lengths are available, it must select a 16-bit encoding. In the few cases where more than one encoding of the same length is available for an instruction, the rules for selecting between them are instruction-specific and are part of the instruction description.

I<sub>BWNR</sub>

Syntax options exist to override the normal instruction selection rules and ensure that a particular encoding is selected. These are useful when disassembling code, to ensure that subsequent assembly produces the original code, and in some other situations.

## C1.2.6 Pseudocode describing how the instruction operates

I<sub>RTDZ</sub>

Each instruction description includes pseudocode that provides a precise description of what the instruction does.

I<sub>LRFZ</sub>

In the instruction pseudocode, instruction fields are referred to by the names shown in the encoding diagram for the instruction.

R<sub>NLPM</sub>

Where the pseudocode describes UNPREDICTABLE behavior the constraints on that behavior are described in the Operation section.

I<sub>BNVW</sub>

Pseudocode does not describe the ordering requirements when an instruction generates multiple memory accesses.

R<sub>CRWM</sub>

Pseudocode describes the exact rules when an UNDEFINED instruction fails its [Condition code check](#).

In such cases, the UNDEFINED pseudocode statement lies inside the if `ConditionPassed()` then ... structure, either directly or in the `EncodingSpecificOperations()` function call, and so the pseudocode indicates that the instruction executes as a NOP.

I<sub>MZKZ</sub>

Pseudocode does not describe the exact ordering requirements when a single floating-point instruction generates more than one floating-point exception and one or more of those floating-point exceptions is trapped.

**I<sub>JMFG</sub>** An exception can be taken during execution of the pseudocode for an instruction, either explicitly as a result of the execution of a pseudocode function, or implicitly, for example if an interrupt is taken during execution of an [LDM](#) instruction. If this happens, the pseudocode does not describe the extent to which the normal behavior of the instruction occurs.

See also:

[Chapter E1 Arm Pseudocode Definition](#) on page 1193.

[B5.10 Ordering requirements for memory accesses](#) on page 158.

[E1.1.1 General limitations of Arm pseudocode](#) on page 1194.

[C1.3.3 Conditional execution of undefined instructions](#) on page 314.

[B4.12 Priority of Floating-point exceptions relative to other Floating-point exceptions](#) on page 141.

[B3.18 Exception handling](#) on page 87.

[B3.22 Exception return](#) on page 101.

### C1.2.7 Use of labels in UAL instruction syntax

**I<sub>BFJV</sub>** The **UAL** syntax for some instructions includes the label of an instruction or a literal data item that is at a fixed offset from the instruction being specified. The assembler must:

1. Calculate the  $PC$  or  $Align(PC, 4)$  value of the instruction. The  $PC$  value of an instruction is its address plus 4 for a T32 instruction. The  $Align(PC, 4)$  value of an instruction is its  $PC$  value ANDed with  $0xFFFFFFFFC$  to force it to be word-aligned.
2. Calculate the offset from the  $PC$  or  $Align(PC, 4)$  value of the instruction to the address of the labeled instruction or literal data item.
3. Assemble a *PC-relative* encoding of the instruction, that is, one that reads its  $PC$  or  $Align(PC, 4)$  value and adds the calculated offset to form the required address.

**I<sub>TCVF</sub>** For instructions that encode a subtraction operation, if the instruction cannot encode the calculated offset, but can encode minus the calculated offset, the instruction encoding specifies a subtraction of minus the calculated offset.

**R<sub>DLPV</sub>** The following instructions include a label:

- **B** and **BL**.
- **CBNZ** and **CBZ**.
- **LDC**, **LDC2**, **LDR**, **LDRB**, **LDRD**, **LDRH**, **LDRSB**, **LDRSH**, **PLD**, **PLI**, and **VLDR**:
  - When the assembler calculates an offset of 0 for the normal syntax of these instructions, it must assemble an encoding that adds 0 to the  $Align(PC, 4)$  value of the instruction. Encodings that subtract 0 from the  $Align(PC, 4)$  value cannot be specified by the normal syntax.
  - There is an alternative syntax for these instructions that specifies the addition or subtraction and the immediate offset explicitly. In this syntax, the label is replaced by  $[PC, \# +/- <imm>]$ , where:
    - \* **+/-**: Is + or omitted to specify that the immediate offset is to be added to the  $Align(PC, 4)$  value, or - if it is to be subtracted.
    - \* **<imm>**: Is the immediate offset.
  - This alternative syntax makes it possible to assemble the encodings that subtract 0 from the  $Align(PC, 4)$

value, and to disassemble them to a syntax that can be re-assembled correctly.

- ADR:
  - When the assembler calculates an offset of 0 for the normal syntax of this instruction, it must assemble the encoding that adds 0 to the  $\text{Align}(PC, 4)$  value of the instruction. The encoding that subtracts from the  $\text{Align}(PC, 4)$  value cannot be specified by the normal syntax.
  - There is an alternative syntax for this instruction that specifies the addition or subtraction and the **immediate value** explicitly, by writing them as additions `ADD <Rd>, PC, #<imm>` or subtractions `SUB <Rd>, PC, #<imm>`. This alternative syntax makes it possible to assemble the encoding that subtracts 0 from the  $\text{Align}(PC, 4)$  value, and to disassemble it to a syntax that can be re-assembled correctly.

### C1.2.8 Using syntax information

$I_{BJGX}$

For a particular encoding:

- There is usually more than one assembler syntax prototype variant that assembles to it.
- The exact set of prototype variants that assemble to it usually depends on the operands to the instruction, for example the register numbers or immediate constants. As an example, for the `AND (register)` instruction, the syntax `AND R0, R0, R8` selects a 32-bit encoding, but `AND R0, R0, R1` selects a 16-bit encoding.

$I_{HQSS}$

For each instruction encoding that belongs to a target instruction set, an assembler can use the information in the encoding to determine whether it can use that particular encoding to encode the instruction requested by the **UAL** source. If multiple encodings can encode the instruction, then:

- If both a 16-bit encoding and a 32-bit encoding can encode the instruction, the architecturally preferred encoding is the 16-bit encoding. This means that the assembler must use the 16-bit encoding instead of the 32-bit encoding.
- If multiple encodings of the same width can encode the instruction, the assembler syntax indicates the preferred encoding, and how software can select other encodings if required. Each encoding also documents **UAL** syntax that selects it in preference to any other encoding. If no encodings of the target instruction set can encode the instruction requested by the **UAL** source, the assembler normally generates an error that indicates that the instruction is not available in that instruction set.

## C1.3 Conditional execution

**I<sub>XDMQ</sub>** *Conditionally executed* means that the instruction only has its normal effect on the programmers' model operation, memory and coprocessors if the N, Z, C, and V flags in the **APSR** satisfy a condition specified in the instruction. If the flags do not satisfy this condition, the instruction acts as a NOP, that is, execution advances to the next instruction as normal, including any relevant checks for exceptions being taken, but has no other effect.

**I<sub>SPPQ</sub>** Most T32 instructions are unconditional. Conditional execution in T32 code can be achieved using any of the following instructions:

- A 16-bit conditional branch instruction, with a branch range of -256 to +254 bytes. See **B** for details.
- A 32-bit conditional branch instruction, with a branch range of approximately  $\pm 1\text{MB}$ . See **B** for details.
- 16-bit Compare and Branch on Zero and Compare and Branch on Nonzero instructions, with a branch range of +4 to +130 bytes. See **CBNZ**, **CBZ** for details.
- A 16-bit If-Then instruction that makes up to four following instructions conditional. See **IT** for details. The instructions that are made conditional by an IT instruction are called its *IT block*. Instructions in an IT block must either all have the same condition, or some can have one condition, and others can have the inverse condition.

**R<sub>FNBQ</sub>** In T32 instructions, the condition (if it is not **AL**) is encoded in a preceding IT instruction, other than **B**, **CBNZ** and **CBZ**. Some conditional branch instructions do not require a preceding IT instruction, and include a condition code in their encoding.

**I<sub>BDMC</sub>** The following table shows the conditions that are available for conditionally executed instructions.

| cond | Mnemonic extension | Meaning, integer arithmetic  | Meaning, Floating-point arithmetic | APSR condition flags |
|------|--------------------|------------------------------|------------------------------------|----------------------|
| 0000 | EQ                 | Equal                        | Equal                              | Z == 1               |
| 0001 | NE                 | Not equal                    | Not equal, or unordered            | Z == 0               |
| 0010 | CS                 | Carry set                    | Greater than, equal or unordered   | C == 1               |
| 0011 | CC                 | Carry clear                  | Less than                          | C == 0               |
| 0100 | MI                 | Minus, negative              | Less than                          | N == 1               |
| 0101 | PL                 | Plus, positive or zero       | Greater than, equal or unordered   | N == 0               |
| 0110 | VS                 | Overflow                     | Unordered                          | V == 1               |
| 0111 | VC                 | No overflow                  | Not unordered                      | V == 0               |
| 1000 | HI                 | Unsigned higher              | Greater than or unordered          | C == 1 and Z == 0    |
| 1001 | LS                 | Unsigned lower or same       | Less than or equal                 | C == 0 or Z == 1     |
| 1010 | GE                 | Signed greater than or equal | Greater than or equal              | N == V               |
| 1011 | LT                 | Signed less than             | Less than or unordered             | N != V               |
| 1100 | GT                 | Signed greater than          | Greater than                       | Z == 0 and N == V    |
| 1101 | LE                 | Signed less than or equal    | Less than, equal or unordered      | Z == 1 and N != V    |
| 1110 | None (AL)          | Always (unconditional)       | Always (unconditional)             | Any                  |

Unordered means at least one NaN operand.

**HS** (unsigned higher or same) is a synonym for **CS**.

**LO** (unsigned lower) is a synonym for **CC**.

**AL** is an optional mnemonic extension for always, except in **IT** instructions. See **IT** for details.

### C1.3.1 Conditional instructions

**R<sub>WRJS</sub>** The instructions that are made conditional by an **IT** instruction must be written with a condition after the mnemonic. These conditions must match the conditions imposed by the IT instruction.

**I<sub>WVXC</sub>** An example of **R<sub>WRJS</sub>** is:

```

1 ITTEE EQ
2 ADDEQ R0, R1
3 SUBEQ R2, R3
4 ADDNE R4, R5
5 SUBNE R6, R7

```

**R<sub>THGJ</sub>** Some instructions cannot be made conditional by an IT instruction. Some instructions can be conditional if they are the last instruction in the IT block, but not otherwise, see the individual instruction descriptions for details.

**R<sub>TGXF</sub>** If the assembler syntax indicates a conditional branch that correctly matches a preceding **IT** instruction, it is assembled using a branch instruction encoding that does not include a condition field.

See also

- [IT](#) instruction

### C1.3.2 Pseudocode details of conditional execution

**R<sub>NMVJ</sub>** The `CurrentCond()` pseudocode function prototype returns a 4-bit condition specifier as follows:

- For the T1 and T3 encodings of the Branch instruction, it returns the 4-bit `cond` field of the encoding.
- For all other T32 instructions:
  - If `ITSTATE.IT[3:0] != '0000'` it returns `ITSTATE.IT[7:4]`
  - If `ITSTATE.IT[7:0] == '00000000'` it returns `'1110'`
  - Otherwise, execution of the instruction is UNPREDICTABLE.

**R<sub>LTPQ</sub>** The `ConditionPassed()` function calls the `ConditionHolds()` function to determine whether the instruction must be executed.

See also

[C1.3.5 ITSTATE](#) on page 315.

B.

### C1.3.3 Conditional execution of undefined instructions

**R<sub>NPNF</sub>** The conditional execution applies to all instructions. This includes undefined instructions and other instructions that would cause entry to the UsageFault or the UNDEFINSTR UsageFault.

**R<sub>PCJZ</sub>** If such an instruction fails its [condition code check](#) the instruction behaves as a `NOP` and does not cause an `UsageFault`.

### C1.3.4 Interaction of undefined instruction behavior with UNPREDICTABLE or CONSTRAINED UNPREDICTABLE instruction behavior

**R<sub>NZWQ</sub>** If this manual describes an instruction as both:

- UNPREDICTABLE and UNDEFINED, then the instruction is UNPREDICTABLE.
- CONSTRAINED UNPREDICTABLE and UNDEFINED, then the instruction is CONSTRAINED UNPREDICTABLE.

### C1.3.5 ITSTATE

**I<sub>RGFT</sub>** ITSTATE is held in [EPSR.IT](#).  
This register holds the If-Then Execution state bits for the T32 [IT](#) instruction.

**R<sub>QKPG</sub>** [EPSR.IT](#) and ITSTATE divide into two subfields:

IT[7:5]

Holds the *base condition* for the current IT block. The base condition is the top 3 bits of the condition specified by the IT instruction.

This subfield is `0b000` when no IT block is active.

IT[4:0]

Encodes:

\* The size of the IT block. This is the number of instructions that are to be conditionally executed. The size of the block is indicated by the position of the least significant 1 in this field which is bit [4-size of the block].

\* The value of the least significant bit, bit[0], of the condition code for each instruction in the block.

\* Changing the value of the least significant bit of a condition code from 0 to 1 inverts the condition code. For example `cond 0000` is `EQ`, and `cond 0001` is `NE`.

This subfield is `0b000000` when no IT block is active.

**R<sub>FPNH</sub>** When an IT instruction is executed, IT bits[7:0] are set according to the condition in the instruction, and the *Then* and *Else* (`T` and `E`) parameters in the instruction.

**R<sub>XKGZ</sub>** An instruction in an IT block is conditional. The condition used is the current value of IT[7:4]. When an instruction in an IT block completes its execution normally, ITSTATE is advanced by shifting IT bits[4:0] left by 1 bit.

**I<sub>VQJM</sub>** For example:

|       |        | <b>IT[7:5]</b> | <b>IT[4:0]</b> |
|-------|--------|----------------|----------------|
| ITTEE | EQ     | 000            | 00111          |
| ADDEQ | R0, R1 | 000            | 01110          |
| SUBEQ | R2, R3 | 000            | 11100          |
| ADDNE | R4, R5 | 000            | 11000          |
| SUBNE | R6, R7 | 000            | 00000          |

**I<sub>KQBQ</sub>** Instructions that can complete their normal execution by branching are only permitted in an IT block as its last instruction, and so always result in `ITSTATE` advancing to normal execution.

**I<sub>FJLN</sub>** In the following table, *P* represents the base condition or the inverse of the base condition.

|           | <b>IT Bits</b> |            |            |            |            |            |                                        |
|-----------|----------------|------------|------------|------------|------------|------------|----------------------------------------|
|           | <b>[7:5]</b>   | <b>[4]</b> | <b>[3]</b> | <b>[2]</b> | <b>[1]</b> | <b>[0]</b> |                                        |
| cond_base | P1             | P2         | P3         | P4         | 1          |            | Entry point for 4-instruction IT block |
| cond_base | P1             | P2         | P3         | 1          | 0          |            | Entry point for 3-instruction IT block |
| cond_base | P1             | P2         | 1          | 0          | 0          |            | Entry point for 2-instruction IT block |
| cond_base | P1             | 1          | 0          | 0          | 0          |            | Entry point for 1-instruction IT block |
| 000       | 0              | 0          | 0          | 0          | 0          |            | Normal execution, not in an IT block   |

Combinations of the IT bits not shown in this table are reserved.

### C1.3.6 Pseudocode details of ITSTATE operation

**I<sub>JLKP</sub>** `ITAdvance()` describes how `ITSTATE` advances after normal execution.

**I<sub>ZGZL</sub>** `InITBlock()` and `LastInITBlock()` test whether the current instruction is in an IT block, and whether it is the last instruction of an IT block.

### C1.3.7 SVC and ISTATE

**R<sub>TSWQ</sub>** The `ReturnAddress()` for an `SVC` instruction must point to the instruction after the `SVC` instruction and advance `ITSTATE`.

**I<sub>QWZW</sub>** When an `SVC` instruction is escalated to `HardFault` resulting in `lookup` the `ReturnAddress()` is `0xEFFFFFFE`.

See also:

[B3.31 Lookup on page 119.](#)

### C1.3.8 CONSTRAINED UNPREDICTABLE behavior and IT blocks

**R<sub>WVWX</sub>** Branching into an IT block, other than by way of exception return or exit from Debug state, leads to `CONSTRAINED UNPREDICTABLE` behavior. Execution starts from the address that is determined by the branch, but each instruction in the IT block is:



- Executed as if the instruction is not in an IT block, meaning that the instruction is executed unconditionally.
- Executed as if the instruction had passed its [Condition code check](#) within an IT block.
- Executed as a NOP. That is, the instruction [behaves as if](#) it had failed the Condition code check.

**R<sub>CPDC</sub>** For exception returns or Debug state exits that cause [EPSR.IT](#) to be set to a reserved value with a nonzero value in [EPSR.IT](#), the [EPSR.IT](#) bits are forced to 0b00000000.

*Note, Debug state requires Halting debug.*

**R<sub>HVNS</sub>** Exception returns or Debug state exits that set [EPSR.IT](#) to a non-reserved value can occur when the flow of execution returns to a point:

- Outside an IT block, but with the [EPSR.IT](#) bits set to a value other than 0b00000000.
- Inside an IT block, but with a different value of the [EPSR.IT](#) bits than if the IT block had been executed without an exception return or Debug state exit.

In this case the instructions at the target of the exception return or Debug state exit does one of the following:

- Execute as if they passed the Condition code check for the remaining iterations of the [EPSR.IT](#) state machine.
- Execute as NOPs. That is, they [behave as if](#) they failed the Condition code check for the remaining iterations of the [EPSR.IT](#) state machine.

*Note, Debug state requires Halting debug.*

**R<sub>LLDK</sub>** A number of instructions in the architecture are described as being CONstrained UNPREDICTABLE either:

- Anywhere within an IT block.
- As an instruction within an IT block, other than the last instruction within an IT block.

Unless otherwise stated in this reference manual, when these instructions are committed for execution, one of the following occurs:

- An UNDEFINED exception is taken.
- The instructions are executed as if they had passed the [condition code check](#).
- The instructions execute as NOPs, as if they had failed the [condition code check](#).

**I<sub>NJKF</sub>** The behavior might in some implementations vary from instruction to instruction, or between different instances of the same instruction.

**R<sub>BWMN</sub>** Branch instructions or other non-sequential instructions that change the [PC](#) are CONstrained UNPREDICTABLE in an IT block. Where these instructions are not treated as UNDEFINED within an IT block, the remaining iterations of the [EPSR.IT](#) state machine is treated in one of the following ways:

- [EPSR.IT](#) is cleared to 0.
- [EPSR.IT](#) advances for either a sequential or a nonsequential change of the [PC](#) in the same way as it does for instructions that are not CONstrained UNPREDICTABLE that cause a sequential change of the [PC](#).

**I<sub>XHBL</sub>** This behavior does not apply to an instruction that is the last instruction in an IT block.

**R<sub>TMWN</sub>** The instructions that are addressed by the updated [PC](#) does one of the following:

- Execute as if they had passed the [condition code check](#) for the remaining iterations of the [EPSR.IT](#) state machine.

- Execute as `NOPS`. That is, they behave as if they had failed the [condition code check](#) for the remaining iterations of the [EPSR.IT](#) state machine.

**R<sub>KVXD</sub>**

The remaining iterations of the [EPSR.IT](#) state machine behave in one of the following ways:

- The [EPSR.IT](#) state machine advances as if it were in an IT block.
- The [EPSR.IT](#) bits are ignored.
- The [EPSR.IT](#) bits are forced to `0b00000000`.

**R<sub>GZBX</sub>**

Execution of an instruction inside an IT block with `ITSTATE` set to zero, an `ICI` value, or a value that is inconsistent with the IT block is UNPREDICTABLE.

See also:

[B3.5 XPSR, APSR, IPSR, and EPSR](#) on page 59.

[B3.5.2 Execution Program Status Register \(EPSR\)](#) on page 60.

## C1.4 Instruction set encoding information

### C1.4.1 UNDEFINED and UNPREDICTABLE instruction set space

- I<sub>FLRZ</sub>** An attempt to execute an unallocated instruction results in either:
- UNPREDICTABLE behavior. The instruction is described as UNPREDICTABLE.
  - An UNDEFINSTR UsageFault. The instruction is described as UNDEFINED.
  - Unallocated instructions in the NOP hint space behave as NOPS.
- R<sub>KDXB</sub>** An instruction is UNDEFINED if it is declared as UNDEFINED in an instruction description.
- R<sub>XDBQ</sub>** An instruction is UNPREDICTABLE if:
- A bit marked (0) or (1) in the encoding diagram of an instruction is not 0 or 1, respectively, and the pseudocode for that encoding does not indicate that a different special case applies.
  - It is declared as UNPREDICTABLE in an instruction description.
- R<sub>TRHK</sub>** Unless otherwise specified, a T32 instruction that is provided by one or more of the architecture extensions is either UNPREDICTABLE or UNDEFINED in an implementation that does not include those extensions. See the individual instruction descriptions for details.

### C1.4.2 Pseudocode descriptions of operations on general-purpose registers and the PC

- R<sub>HRGP</sub>** In pseudocode, the uses of the [R\[\]](#) function are:
- Reading or writing R0-R12, [SP](#), and [LR](#), using n = 0-12, 13, and 14 respectively.
  - Reading the [PC](#), using n = 15.
- R<sub>CHTM</sub>** The use of [RSPCheck \(\)](#) returns the value of the current [SP](#)

See also:

[R\[\]](#). [RSPCheck \(\)](#)

### C1.4.3 Use of 0b1111 as a register specifier

- R<sub>WMVJ</sub>** All use of the [PC](#) as a named register specifier for a source register that is described as CONSTRAINED UNPREDICTABLE in the pseudocode or in other places in this reference manual does one of the following:
- Cause the instruction to be treated as UNDEFINED.
  - Cause the instruction to be executed as a NOP.
  - Read an UNKNOWN value for the source register that is specified as the [PC](#).
- R<sub>BGJG</sub>** All use of the [PC](#) as a named register specifier for a destination register that is described as CONSTRAINED UNPREDICTABLE in the pseudocode or in other places in this reference manual does one of the following:

- Cause the instruction to be treated as UNDEFINED.
- Cause the instruction to be executed as a NOP.
- Ignore the write.
- Branch to an UNKNOWN location.

|                         |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                          |
|-------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>I<sub>QVWL</sub></b> | The choice between the behavior of the PC as a source or destination register might vary in some implementations from instruction to instruction, or between different instances of the same instruction.                                                                                                                                                                                                                                                                                                                                                                                                                |
| <b>R<sub>LXPR</sub></b> | For instructions that specify two destination registers and if one is specified as the PC, then the other destination register of the pair is UNKNOWN. The CONSTRAINED UNPREDICTABLE behavior for the write to the PC is either to ignore the write or to branch to an UNKNOWN location.                                                                                                                                                                                                                                                                                                                                 |
| <b>R<sub>DRSS</sub></b> | An instruction that specifies the PC as a Base register and specifies a base register writeback is CONSTRAINED UNPREDICTABLE and behaves as if the PC is both the source and destination register.                                                                                                                                                                                                                                                                                                                                                                                                                       |
| <b>R<sub>XLVX</sub></b> | For instructions that affect any or all of APSR.{N, Z, C, V} or APSR.GE when the register specifier is not the PC, any flags that are affected by an instruction that is CONSTRAINED UNPREDICTABLE become UNKNOWN.                                                                                                                                                                                                                                                                                                                                                                                                       |
| <b>R<sub>JFGT</sub></b> | For MRC instructions that use the PC as the destination register descriptor (and therefore target APSR.{N, Z, C, V}) and where these instructions are described as being CONSTRAINED UNPREDICTABLE the status of the flags becomes UNKNOWN.                                                                                                                                                                                                                                                                                                                                                                              |
| <b>R<sub>XPBT</sub></b> | Multi-access instructions that load the PC from Device memory are CONSTRAINED UNPREDICTABLE and one of the following behaviors occurs: <ul style="list-style-type: none"> <li>• The instruction loads the PC from the memory location as if the memory location had the Normal Non-cacheable attribute.</li> <li>• The instruction generates a MemManage fault.</li> </ul>                                                                                                                                                                                                                                               |
| <b>R<sub>RTKM</sub></b> | All unallocated or reserved values of fields with allocated values within the memory-mapped registers that are described in this reference manual behave, unless otherwise stated in the register description, in one of the following ways: <ul style="list-style-type: none"> <li>• The encoding maps onto any of the allocated values, but otherwise does not cause CONSTRAINED UNPREDICTABLE behavior.</li> <li>• The encoding causes effects that could be achieved by a combination of more than one of the allocated encodings.</li> <li>• The encoding causes the field to have no functional effect.</li> </ul> |
| <b>I<sub>QXNP</sub></b> | When a value of 0b1111 is permitted as a register specifier, as indicated in the individual instruction descriptions, a variety of meanings is possible. For register reads, these meanings are: <ul style="list-style-type: none"> <li>• Read the PC value, that is, the address of the current instruction + 4. The base register of the table branch instructions TBB and TBH can be the PC. This enables branch tables to be placed in memory immediately after the instruction. (Some instructions read the PC value implicitly, without the use of a register specifier,</li> </ul>                                |

for example the conditional branch instruction `B<cond>`.)

- Read the word-aligned **PC** value, that is, the address of the current instruction + 4, with bits [1:0] forced to zero. The base register of `LDC`, `LDR`, `LDRB`, `LDRD` (pre-indexed, no write-back), `LDRH`, `LDRSB`, and `LDRSH` instructions can be the word-aligned **PC**. This enables PC-relative data addressing. In addition, some encodings of the `ADD` and `SUB` instructions permit their source registers to be `0b1111` for the same purpose.
- Read zero. This is done in some cases when one instruction is a special case of another, more general instruction, but with one operand zero. In these cases, the instructions are listed on separate pages, with a special case in the pseudocode for the more general instruction cross-referencing the other page. An example of this is the descriptions of `MOV (register)` and `ORR (register)`.

**I<sub>KVHQ</sub>**

When a value of `0b1111` is permitted as a register specifier, as indicated in the individual instruction descriptions, a variety of meanings is possible. For register writes, these meanings are:

- The **PC** can be specified as the destination register of an `LDR` instruction. This is done by encoding `Rt` as `0b1111`. The loaded value is treated as an address, and the effect of execution is a branch to that address. `bit[0]` of the loaded value selects the Execution state after the branch and must have the value 1.
- Discard the result of a calculation. This is done in some cases when one instruction is a special case of another, more general instruction, but with the result discarded. In these cases, the instructions are listed on separate pages, with a special case in the pseudocode for the more general instruction cross-referencing the other page. An example of this is the descriptions of `TST (register)` and `AND (register)`.
- If the destination register specifier of an `LDRB`, `LDRH`, `LDRSB`, or `LDRSH` instruction is `0b1111`, the instruction is a memory hint instead of a load operation.
- If the destination register specifier of an `MRC` instruction is `0b1111`, `bits[31:28]` of the value transferred from the coprocessor are written to the `N`, `Z`, `C`, and `V` flags in the `APSR`, and `bits[27:0]` are discarded.

#### C1.4.4 Use of `0b1101` as a register specifier

##### **SP[1:0] definition**

**R<sub>DSDB</sub>**

Bits [1:0] of **SP** must be treated as *SBZP* (Should Be Zero or Preserved). Writing a nonzero value to bits [1:0] results in UNPREDICTABLE behavior. Reading bits [1:0] returns zero.

##### **32-bit T32 instruction support for SP**

**R<sub>SKNR</sub>**

Use of the **SP** in T32 instructions and 16-bit data processing instructions is restricted to the following cases:

- **SP** as the source or destination register of a `MOV` instruction. Only register to register transfers without shifts are supported, with no flag setting:

|   |                  |                     |  |
|---|------------------|---------------------|--|
| 1 | <code>MOV</code> | <code>SP, Rm</code> |  |
| 2 | <code>MOV</code> | <code>Rn, SP</code> |  |

- Adjusting **SP** up or down by a multiple of its alignment:

|   |                     |                                    |                         |
|---|---------------------|------------------------------------|-------------------------|
| 1 | <code>ADD{W}</code> | <code>SP, SP, #N</code>            | ; For N a multiple of 4 |
| 2 | <code>SUB{W}</code> | <code>SP, SP, #N</code>            | ; For N a multiple of 4 |
| 3 | <code>ADD</code>    | <code>SP, SP, Rm, LSL #shft</code> | ; For shft=0,1,2,3      |
| 4 | <code>SUB</code>    | <code>SP, SP, Rm, LSL #shft</code> | ; For shft=0,1,2,3      |

- **SP** as a base register, `Rn`, of any load or store instruction. This supports **SP**-based addressing for load, store,

- or memory hint instructions, with positive or negative offsets, with and without write-back.
- **SP** as the first operand, **Rn**, in any `ADD{S}`, `CMN`, `CMP`, or `SUB{S}` instruction. The add and subtract instructions support **SP**-based address generation, with the address going into a general-purpose register. `CMN` and `CMP` can check the stack pointer.
- **SP** as the transferred register, **Rt**, in any `LDR` or `STR` instruction.
- **SP** as the address in a `POP` or `PUSH` instruction.

**R<sub>MRNT</sub>** Where an instruction states that the **SP** is UNPREDICTABLE and **SP** is used:

- The value that is read or written from or to the **SP** is UNKNOWN.
- The instruction is permitted to be treated as UNDEFINED.
- If the **SP** is being written, it is UNKNOWN whether a stack-limit check is applied.

### C1.4.5 16-bit T32 instruction support for SP

**R<sub>STHZ</sub>** Arm deprecates any other use of the **SP** in T16 instructions. This affects the high register forms of `CMP` and `ADD`, where Arm deprecates the use of **SP** as **Rm**.

### C1.4.6 Branching

**I<sub>PVGL</sub>** Writing an address to the **PC** causes either a simple branch to that address or an *interworking* branch.

**R<sub>CCVD</sub>** A simple branch is performed by `BranchWritePC()`.

**R<sub>XMGH</sub>** An interworking branch is performed by `BXWritePC()`.

**R<sub>CWSL</sub>** Branching can occur in cases where `0b1111` is not a register specifier. In these cases, instructions write the **PC** either:

- Implicitly, for example, `b<cond>`.
- By using a register mask rather than a register specifier, for example `LDM`.

**I<sub>FLZZ</sub>** The address to branch to can be:

- A loaded value, for example `LDM`.
- A register value, for example `BX`.
- The result of a calculation, for example `TBB` or `TBH`.

**R<sub>WQBX</sub>** The following table summarizes the branch instructions in the T32 instruction set.

| Instruction                                                  | See                                         | Range, T32    |
|--------------------------------------------------------------|---------------------------------------------|---------------|
| Branch to target address                                     | <a href="#">B</a>                           | ±16MB         |
| Compare and Branch on Nonzero,<br>Compare and Branch on Zero | <a href="#">CBNZ</a> , <a href="#">CBZ</a>  | 0-126 bytes   |
| Call a subroutine                                            | <a href="#">BL</a>                          | ±16MB         |
| Call a subroutine, optionally change Security state          | <a href="#">BLX</a> , <a href="#">BLXNS</a> | Any           |
| Branch to target address, change to Non-secure state         | <a href="#">BX</a> , <a href="#">BXNS</a>   | Any           |
| Table Branch (byte offsets)                                  | <a href="#">TBB</a> , <a href="#">TBH</a>   | 0-510 bytes   |
| Table Branch (halfword offsets)                              |                                             | 0-31070 bytes |

**R<sub>GJML</sub>** Branches to loaded and calculated addresses can be performed by [LDR](#), [LDM](#) and data-processing instructions.

**R<sub>TPTF</sub>** A load instruction that targets the [PC](#) behaves as a branch instruction.

### C1.4.7 Instruction set, interworking and interstating support

**R<sub>LBQC</sub>** The following instructions are [Interworking](#) branches:

- [BX](#) and [BLX](#).
- [POP \(multiple registers\)](#) and all forms of [LDM](#), when the register list includes the [PC](#).
- [LDR \(immediate\)](#), [LDR \(literal\)](#), and [LDR \(register\)](#), with <Rt> equal to the [PC](#).

**R<sub>WRZR</sub>** The value of bit[0] of an interworking branch instruction is not stored in the [PC](#). Bit[0] of an interworking branch instruction sets [EPSR.T](#). If [EPSR.T](#) is cleared to 0 an INVSTATE UsageFault or HardFault is generated on the next instruction the PE attempts to execute.

*Note, requires M for INVSTATE UsageFault.*

**R<sub>GLPL</sub>** The following instructions are *interstating branches*:

- [BXNS](#) and [BLXNS](#).

*The extension requirements are - S.*

**R<sub>GJMJ</sub>** When an interstating branch is executed in Secure state, bit[0] of the target address indicates the target Security state:

**0:** The target Security state is Non-secure state.

**1:** The target Security state is Secure state.

The value of bit[0] of an interstating branch instruction is not stored in the [PC](#).

*The extension requirements are - S.*

**R<sub>WNSX</sub>** Interstating branches are UNDEFINED when executing in Non-secure state.

*The extension requirements are - S.*

See also:

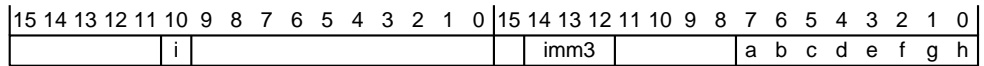
[C1.1 Instruction set on page 306.](#)

[BXWritePC\(\)](#).

[B3.15 Security state transitions on page 82.](#)

## C1.5 Modified immediate constants

**R<sub>JVCL</sub>** The encoding of modified immediate constants in T32 instructions is:



**R<sub>TCLZ</sub>** The table shows the range of modified immediate constants available in T32 data processing instructions, and how they are encoded in the a, b, c, d, e, f, g, h, i, and imm3 fields in the instruction.

| i:imm3:a | <const>                                 |          |          |          | Carry flag set |
|----------|-----------------------------------------|----------|----------|----------|----------------|
| 0000x    | 00000000                                | 00000000 | 00000000 | abcdefgh | No             |
| 0001x    | 00000000                                | abcdefgh | 00000000 | abcdefgh | No             |
| 0010x    | abcdefgh                                | 00000000 | abcdefgh | 00000000 | No             |
| 0011x    | abcdefgh                                | abcdefgh | abcdefgh | abcdefgh | No             |
| 01000    | 1bcdefgh                                | 00000000 | 00000000 | 00000000 | Yes, to 1      |
| 01001    | 01bcdefg                                | h0000000 | 00000000 | 00000000 | Yes, to 0      |
| 01010    | 001bcdef                                | gh000000 | 00000000 | 00000000 | Yes, to 0      |
| 01011    | 0001bcde                                | fgh00000 | 00000000 | 00000000 | Yes, to 0      |
| -        |                                         |          |          |          | Yes, to 0      |
| -        | 8-bit values shifted to other positions |          |          |          |                |
| -        |                                         |          |          |          |                |
| 11101    | 00000000                                | 00000000 | 000001bc | defgh000 | Yes, to 0      |
| 11110    | 00000000                                | 00000000 | 0000001b | cdefgh00 | Yes, to 0      |
| 11111    | 00000000                                | 00000000 | 00000001 | bcdefgh0 | Yes, to 0      |

This table shows the immediate constant value in binary form, to relate *abcdefgh* to the encoding diagram. In assembly syntax, the **immediate value** is specified as a decimal integer by default.

The setting of the Carry flag will only apply if a logical operation with a modified immediate constant can set the flags.

Where *i:imm3:a* is 0001x, 0010x or 0011x the instruction will be UNPREDICTABLE if *abcdefgh* == 0b00000000.

### C1.5.1 Operation of modified immediate constants

**R<sub>TLLFG</sub>** [T32ExpandImm\(\)](#) and [T32T32ExpandImm\\_C\(\)](#) describe the operation of modified immediate constants.

**R<sub>PHBG</sub>** The operation of modified immediate constants are UNPREDICTABLE where both:

- *hw2*[7:0] == 0b00000000.
- *hw1*[10] == 0 and either:
  - *hw2* [14:12] == 0b001.
  - *hw2* [14:12] == 0b010.
  - *hw2* [14:12] == 0b011.



## C1.6 NOP-compatible hint instructions

**I<sub>BJRT</sub>** A hint instruction only provides an indication to the PE. It is not required that the PE perform an operation on a hint instruction.

**R<sub>VXQV</sub>** A NOP-compatible hint instruction either:

- Acts as a `NOP` (No Operation) instruction.
- Performs some `IMPLEMENTATION DEFINED` behavior.

**R<sub>DBNQ</sub>** A PE without the Main Extension only supports the 16-bit encodings of the Armv8-M NOP-compatible hint instructions.

*The extension requirements are - **M**.*

**R<sub>DJQL</sub>** A PE with the Main Extension supports both the 16-bit and the 32-bit encodings of the Armv8-M NOP-compatible hint instructions.

*The extension requirements are - **M**.*

See also

[Hints, T16.](#)

[Hints, T32.](#)

## C1.7 SBZ or SBO fields in instructions

$I_{PWBN}$  Many of the instructions have (0) or (1) in the instruction decode to indicate *Should-Be-Zero*, SBZ, or *Should-Be-One*, SBO.

$R_{CKJK}$  If the instruction bit pattern of an instruction is executed with these fields not having the *should-be* values, one of the following must occur:

- The instruction is UNDEFINED.
- The instruction executes as a NOP.
- The instruction operates as if the bit had the *should-be* value.
- Any destination registers of the instruction become UNKNOWN.

The exceptions to this rule are:

- LDM, LDMIA, LDMFD.
- LDMDB, LDMEA.
- LDR (immediate).
- LDRB (immediate).
- LDRD (immediate).
- LDRH (immediate).
- LDRSB (literal).
- LDRSH (literal).
- POP (multiple registers).
- PUSH (multiple registers).
- SDIV.
- STM, STMIA, STMEA.
- STMDB, STMFD.
- UDIV.

## Chapter C2

# Instruction Specification

This chapter specifies the Armv8-M instruction set. It contains the following sections:

[Top level T32 instruction set encoding.](#)

[16-bit T32 instruction encoding.](#)

[32-bit T32 instruction encoding.](#)

[Alphabetical list of instructions.](#)



## C2.2 16-bit T32 instruction encoding

This section describes the encoding of the 16-bit T32 instruction encoding group. This section is decoded from [Top level T32 instruction set encoding](#).

### Note

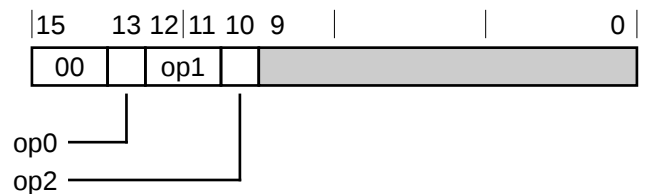
In the decode tables in this section, an entry of - for a field value means the value of the field does not affect the decoding.



| Decode fields | Decode group or instruction page                    |
|---------------|-----------------------------------------------------|
| <b>op0</b>    |                                                     |
| 00xxxx        | Shift (immediate), add, subtract, move, and compare |
| 010000        | Data-processing (two low registers)                 |
| 010001        | Special data instructions and branch and exchange   |
| 01001x        | LDR (literal) - T1 variant                          |
| 0101xx        | Load/store (register offset)                        |
| 011xxx        | Load/store word/byte (immediate offset)             |
| 1000xx        | Load/store halfword (immediate offset)              |
| 1001xx        | Load/store (SP-relative)                            |
| 1010xx        | Add PC/SP (immediate)                               |
| 1011xx        | Miscellaneous 16-bit instructions                   |
| 1100xx        | Load/store multiple                                 |
| 1101xx        | Conditional branch, and Supervisor Call             |

### C2.2.1 Shift (immediate), add, subtract, move, and compare

This section describes the encoding of the Shift (immediate), add, subtract, move, and compare group. The encodings in this section are decoded from [16-bit T32 instruction encoding](#).



| Decode fields |       |     | Decode group or instruction page                              |
|---------------|-------|-----|---------------------------------------------------------------|
| op0           | op1   | op2 |                                                               |
| 0             | 11    | 0   | Add, subtract (three low registers)                           |
| 0             | 11    | 1   | Add, subtract (two low registers and immediate)               |
| 0             | != 11 | -   | MOV (register) - T2 variant                                   |
| 1             | -     | -   | Add, subtract, compare, move (one low register and immediate) |

### Add, subtract (three low registers)

This section describes the encoding of the Add, subtract (three low registers) instruction class. The encodings in this section are decoded from [Shift \(immediate\)](#), [add, subtract, move, and compare](#).

|    |    |    |    |    |    |   |    |    |    |   |   |   |
|----|----|----|----|----|----|---|----|----|----|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8  | 6  | 5  | 3 | 2 | 0 |
| 0  | 0  | 0  | 1  | 1  | 0  | S | Rm | Rn | Rd |   |   |   |

| Decode fields | Instruction page               |
|---------------|--------------------------------|
| S             |                                |
| 0             | <a href="#">ADD (register)</a> |
| 1             | <a href="#">SUB (register)</a> |

### Add, subtract (two low registers and immediate)

This section describes the encoding of the Add, subtract (two low registers and immediate) instruction class. The encodings in this section are decoded from [Shift \(immediate\)](#), [add, subtract, move, and compare](#).

|    |    |    |    |    |    |   |      |    |    |   |   |   |
|----|----|----|----|----|----|---|------|----|----|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8    | 6  | 5  | 3 | 2 | 0 |
| 0  | 0  | 0  | 1  | 1  | 1  | S | imm3 | Rn | Rd |   |   |   |

| Decode fields | Instruction page                |
|---------------|---------------------------------|
| S             |                                 |
| 0             | <a href="#">ADD (immediate)</a> |
| 1             | <a href="#">SUB (immediate)</a> |

### Add, subtract, compare, move (one low register and immediate)

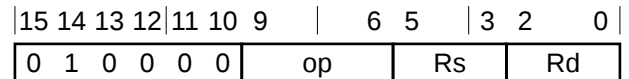
This section describes the encoding of the Add, subtract, compare, move (one low register and immediate) instruction class. The encodings in this section are decoded from [Shift \(immediate\)](#), [add, subtract, move, and compare](#).

|    |    |    |    |    |    |   |   |  |  |      |   |
|----|----|----|----|----|----|---|---|--|--|------|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 8 | 7 |  |  |      | 0 |
| 0  | 0  | 1  | op | Rd |    |   |   |  |  | imm8 |   |

| Decode fields | Instruction page                |
|---------------|---------------------------------|
| op            |                                 |
| 00            | <a href="#">MOV (immediate)</a> |
| 01            | <a href="#">CMP (immediate)</a> |
| 10            | <a href="#">ADD (immediate)</a> |
| 11            | <a href="#">SUB (immediate)</a> |

### C2.2.2 Data-processing (two low registers)

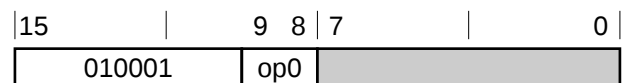
This section describes the encoding of the Data-processing (two low registers) instruction class. The encodings in this section are decoded from [16-bit T32 instruction encoding](#).



| Decode fields | Instruction page                                                       |
|---------------|------------------------------------------------------------------------|
| <b>op</b>     |                                                                        |
| 0000          | <a href="#">AND (register)</a>                                         |
| 0001          | <a href="#">EOR (register)</a>                                         |
| 0010          | MOV, MOVS (register-shifted register) - Logical shift left variant     |
| 0011          | MOV, MOVS (register-shifted register) - Logical shift right variant    |
| 0100          | MOV, MOVS (register-shifted register) - Arithmetic shift right variant |
| 0101          | <a href="#">ADC (register)</a>                                         |
| 0110          | <a href="#">SBC (register)</a>                                         |
| 0111          | MOV, MOVS (register-shifted register) - Rotate right variant           |
| 1000          | <a href="#">TST (register)</a>                                         |
| 1001          | <a href="#">RSB (immediate)</a>                                        |
| 1010          | <a href="#">CMP (register)</a>                                         |
| 1011          | <a href="#">CMN (register)</a>                                         |
| 1100          | <a href="#">ORR (register)</a>                                         |
| 1101          | <a href="#">MUL</a>                                                    |
| 1110          | <a href="#">BIC (register)</a>                                         |
| 1111          | <a href="#">MVN (register)</a>                                         |

### C2.2.3 Special data instructions and branch and exchange

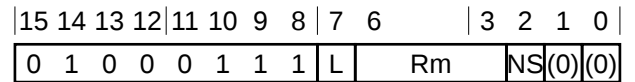
This section describes the encoding of the Special data instructions and branch and exchange group. The encodings in this section are decoded from [16-bit T32 instruction encoding](#).



| Decode fields | Decode group or instruction page                                  |
|---------------|-------------------------------------------------------------------|
| <b>op0</b>    |                                                                   |
| 11            | <a href="#">Branch and exchange</a>                               |
| != 11         | <a href="#">Add, subtract, compare, move (two high registers)</a> |

#### Branch and exchange

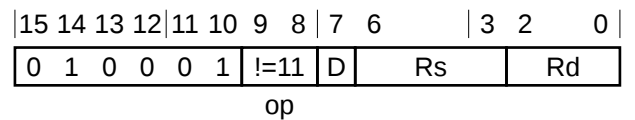
This section describes the encoding of the Branch and exchange instruction class. The encodings in this section are decoded from [Special data instructions and branch and exchange](#).



| Decode fields | Instruction page           |
|---------------|----------------------------|
| <b>L</b>      |                            |
| 0             | <a href="#">BX, BXNS</a>   |
| 1             | <a href="#">BLX, BLXNS</a> |

### Add, subtract, compare, move (two high registers)

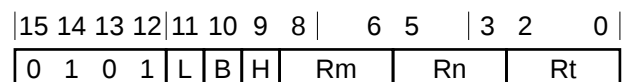
This section describes the encoding of the Add, subtract, compare, move (two high registers) instruction class. The encodings in this section are decoded from [Special data instructions and branch and exchange](#).



| Decode Fields |         |         | Instruction page                            |
|---------------|---------|---------|---------------------------------------------|
| op            | D:Rd    | RS      |                                             |
| 00            | != 1101 | != 1101 | <a href="#">ADD (register)</a>              |
| 00            | -       | 1101    | <a href="#">ADD (SP plus register) - T1</a> |
| 00            | 1101    | != 1101 | <a href="#">ADD (SP plus register) - T2</a> |
| 01            | -       | -       | <a href="#">CMP (register)</a>              |
| 10            | -       | -       | <a href="#">MOV (register)</a>              |

### C2.2.4 Load/store (register offset)

This section describes the encoding of the Load/store (register offset) instruction class. The encodings in this section are decoded from [16-bit T32 instruction encoding](#).

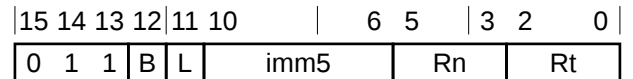


| Decode Fields |   |   | Instruction page                 |
|---------------|---|---|----------------------------------|
| L             | B | H |                                  |
| 0             | 0 | 0 | <a href="#">STR (register)</a>   |
| 0             | 0 | 1 | <a href="#">STRH (register)</a>  |
| 0             | 1 | 0 | <a href="#">STRB (register)</a>  |
| 0             | 1 | 1 | <a href="#">LDRSB (register)</a> |
| 1             | 0 | 0 | <a href="#">LDR (register)</a>   |
| 1             | 0 | 1 | <a href="#">LDRH (register)</a>  |
| 1             | 1 | 0 | <a href="#">LDRB (register)</a>  |
| 1             | 1 | 1 | <a href="#">LDRSH (register)</a> |



### C2.2.5 Load/store word/byte (immediate offset)

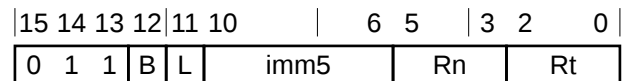
This section describes the encoding of the Load/store word/byte (immediate offset) instruction class. The encodings in this section are decoded from [16-bit T32 instruction encoding](#).



| Decode fields |   | Instruction page                 |
|---------------|---|----------------------------------|
| B             | L |                                  |
| 0             | 0 | <a href="#">STR (immediate)</a>  |
| 0             | 1 | <a href="#">LDR (immediate)</a>  |
| 1             | 0 | <a href="#">STRB (immediate)</a> |
| 1             | 1 | <a href="#">LDRB (immediate)</a> |

### C2.2.6 Load/store halfword (immediate offset)

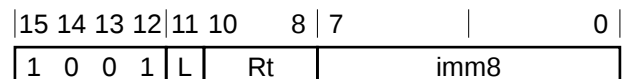
This section describes the encoding of the Load/store halfword (immediate offset) instruction class. The encodings in this section are decoded from [16-bit T32 instruction encoding](#).



| Decode fields |  | Instruction page                 |
|---------------|--|----------------------------------|
| L             |  |                                  |
| 0             |  | <a href="#">STRH (immediate)</a> |
| 1             |  | <a href="#">LDRH (immediate)</a> |

### C2.2.7 Load/store (SP-relative)

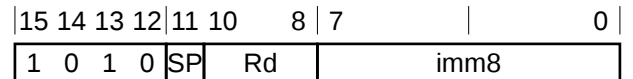
This section describes the encoding of the Load/store (SP-relative) instruction class. The encodings in this section are decoded from [16-bit T32 instruction encoding](#).



| Decode fields |  | Instruction page                |
|---------------|--|---------------------------------|
| L             |  |                                 |
| 0             |  | <a href="#">STR (immediate)</a> |
| 1             |  | <a href="#">LDR (immediate)</a> |

### C2.2.8 Add PC/SP (immediate)

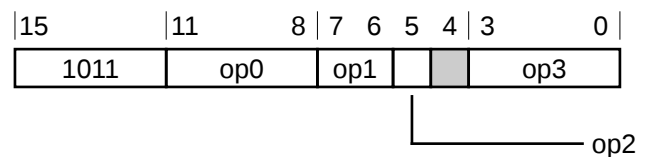
This section describes the encoding of the Add PC/SP (immediate) instruction class. The encodings in this section are decoded from [16-bit T32 instruction encoding](#).



| Decode fields | Instruction page                        |
|---------------|-----------------------------------------|
| <b>SP</b>     |                                         |
| 0             | <a href="#">ADR</a>                     |
| 1             | <a href="#">ADD (SP plus immediate)</a> |

### C2.2.9 Miscellaneous 16-bit instructions

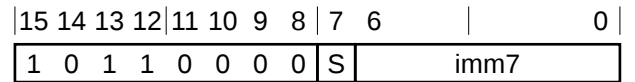
This section describes the encoding of the Miscellaneous 16-bit instructions group. The encodings in this section are decoded from [16-bit T32 instruction encoding](#).



| Decode fields |       |     |         | Decode group or instruction page      |
|---------------|-------|-----|---------|---------------------------------------|
| op0           | op1   | op2 | op3     |                                       |
| 0000          | –     | –   | –       | <a href="#">Adjust SP (immediate)</a> |
| 0010          | –     | –   | –       | <a href="#">Extend</a>                |
| 0110          | 00    | –   | –       | Unallocated.                          |
| 0110          | 01    | 0   | –       | Unallocated.                          |
| 0110          | 01    | 1   | –       | <a href="#">CPS</a>                   |
| 0110          | 1x    | –   | –       | Unallocated.                          |
| 0111          | –     | –   | –       | Unallocated.                          |
| 1000          | –     | –   | –       | Unallocated.                          |
| 1010          | 10    | –   | –       | Unallocated.                          |
| 1010          | != 10 | –   | –       | <a href="#">Reverse bytes</a>         |
| 1110          | –     | –   | –       | <a href="#">BKPT</a>                  |
| 1111          | –     | –   | 0000    | <a href="#">Hints</a>                 |
| 1111          | –     | –   | != 0000 | <a href="#">IT</a>                    |
| x0x1          | –     | –   | –       | <a href="#">CBNZ, CBZ</a>             |
| x10x          | –     | –   | –       | <a href="#">Push and Pop</a>          |

### Adjust SP (immediate)

This section describes the encoding of the Adjust SP (immediate) instruction class. The encodings in this section are decoded from [Miscellaneous 16-bit instructions](#).



| Decode fields |  | Instruction page         |
|---------------|--|--------------------------|
| <b>S</b>      |  |                          |
| 0             |  | ADD (SP plus immediate)  |
| 1             |  | SUB (SP minus immediate) |

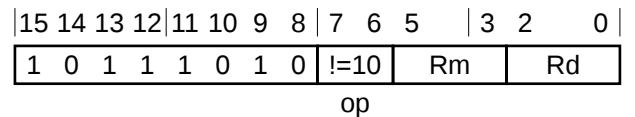
## Extend

This section describes the encoding of the Extend instruction class. The encodings in this section are decoded from [Miscellaneous 16-bit instructions](#).

| Decode fields |          | Instruction page |
|---------------|----------|------------------|
| <b>U</b>      | <b>B</b> |                  |
| 0             | 0        | SXTH             |
| 0             | 1        | SXTB             |
| 1             | 0        | UXTH             |
| 1             | 1        | UXTB             |

## Reverse bytes

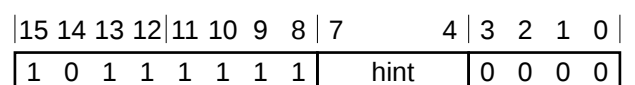
This section describes the encoding of the Reverse bytes instruction class. The encodings in this section are decoded from [Miscellaneous 16-bit instructions](#).



| Decode fields |  | Instruction page |
|---------------|--|------------------|
| <b>op</b>     |  |                  |
| 00            |  | REV              |
| 01            |  | REV16            |
| 11            |  | REVSH            |

## Hints

This section describes the encoding of the Hints instruction class. The encodings in this section are decoded from [Miscellaneous 16-bit instructions](#).



| Decode fields | Instruction page                                |
|---------------|-------------------------------------------------|
| <b>hint</b>   |                                                 |
| 0000          | <a href="#">NOP</a>                             |
| 0001          | <a href="#">YIELD</a>                           |
| 0010          | <a href="#">WFE</a>                             |
| 0011          | <a href="#">WFI</a>                             |
| 0100          | <a href="#">SEV</a>                             |
| 0101          | Reserved hint, behaves as <a href="#">NOP</a> . |
| 011x          | Reserved hint, behaves as <a href="#">NOP</a> . |
| 1xxx          | Reserved hint, behaves as <a href="#">NOP</a> . |

## Push and Pop

This section describes the encoding of the Push and Pop instruction class. The encodings in this section are decoded from [Miscellaneous 16-bit instructions](#).

|    |    |    |    |    |    |   |   |   |               |  |  |   |
|----|----|----|----|----|----|---|---|---|---------------|--|--|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 |               |  |  | 0 |
| 1  | 0  | 1  | 1  | L  | 1  | 0 | P |   | register_list |  |  |   |

| Decode fields | Instruction page                  |
|---------------|-----------------------------------|
| <b>L</b>      |                                   |
| 0             | <a href="#">STMDB, STMFD</a>      |
| 1             | <a href="#">LDM, LDMIA, LDMFD</a> |

### C2.2.10 Load/store multiple

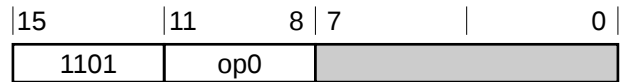
This section describes the encoding of the Load/store multiple instruction class. The encodings in this section are decoded from [16-bit T32 instruction encoding](#).

|    |    |    |    |    |    |   |   |               |  |  |   |
|----|----|----|----|----|----|---|---|---------------|--|--|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 8 | 7 |               |  |  | 0 |
| 1  | 1  | 0  | 0  | L  | Rn |   |   | register_list |  |  |   |

| Decode fields | Instruction page                  |
|---------------|-----------------------------------|
| <b>L</b>      |                                   |
| 0             | <a href="#">STM, STMIA, STMEA</a> |
| 1             | <a href="#">LDM, LDMIA, LDMFD</a> |

### C2.2.11 Conditional branch, and Supervisor Call

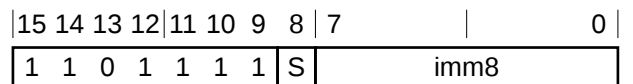
This section describes the encoding of the Conditional branch, and Supervisor Call group. The encodings in this section are decoded from [16-bit T32 instruction encoding](#).



| Decode fields | Decode group or instruction page     |
|---------------|--------------------------------------|
| op0           |                                      |
| 111x          | <a href="#">Exception generation</a> |
| != 111x       | <a href="#">B - T1 variant</a>       |

### C2.2.11.1 Exception generation

This section describes the encoding of the Exception generation instruction class. The encodings in this section are decoded from [Conditional branch, and Supervisor Call](#).



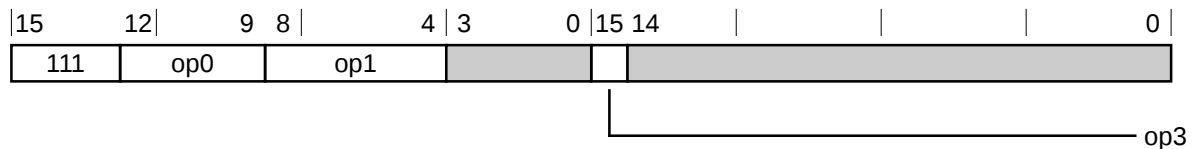
| Decode fields | Instruction page    |
|---------------|---------------------|
| S             |                     |
| 0             | <a href="#">UDF</a> |
| 1             | <a href="#">SVC</a> |

## C2.3 32-bit T32 instruction encoding

This section describes the encoding of the 32-bit T32 instruction encoding group. This section is decoded from [Top level T32 instruction set encoding](#).

### Note

In the decode tables in this section, an entry of - for a field value means the value of the field does not affect the decoding.



| Decode fields |           | Decode group or instruction page |                                                                       |
|---------------|-----------|----------------------------------|-----------------------------------------------------------------------|
| op0           | op1       | op3                              |                                                                       |
| x11x          | -         | -                                | Coprocessor and floating-point instructions                           |
| 0100          | -         | -                                | Load/store (multiple, dual, exclusive, acquire-release), table branch |
| 0101          | -         | -                                | Data-processing (shifted register)                                    |
| 10xx          | -         | 1                                | Branches and miscellaneous control                                    |
| 10x0          | -         | 0                                | Data-processing (modified immediate)                                  |
| 10x1          | -         | 0                                | Data-processing (plain binary immediate)                              |
| 1100          | 1xxxx0    | -                                | Unallocated.                                                          |
| 1100          | != 1xxxx0 | -                                | Load/store single                                                     |
| 1101          | 0xxxxx    | -                                | Data-processing (register)                                            |
| 1101          | 10xxxx    | -                                | Multiply, multiply accumulate, and absolute difference                |
| 1101          | 11xxxx    | -                                | Long multiply and divide                                              |

### C2.3.1 Load/store (multiple, dual, exclusive, acquire-release), table branch

This section describes the encoding of the Load/store (multiple, dual, exclusive, acquire-release), table branch group. The encodings in this section are decoded from [32-bit T32 instruction encoding](#).



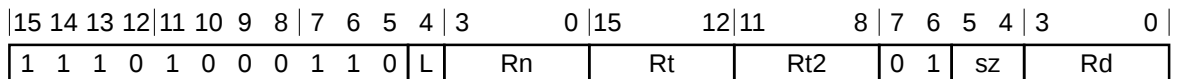
| Decode fields |     | Decode group or instruction page |                                                                |
|---------------|-----|----------------------------------|----------------------------------------------------------------|
| op0           | op1 |                                  |                                                                |
| -             | 0x  |                                  | Load/store multiple                                            |
| 0             | 10  |                                  | Load/store exclusive, load-acquire/store-release, table branch |
| 0             | 11  |                                  | Load/store dual (post-indexed)                                 |
| 1             | 10  |                                  | Load/store dual (literal and immediate)                        |
| 1             | 11  |                                  | Load/store dual (pre-indexed), secure gateway                  |



| Decode fields | Instruction page |
|---------------|------------------|
| <b>L</b>      |                  |
| 0             | STREX            |
| 1             | LDREX            |

### Load/store exclusive byte/half/dual

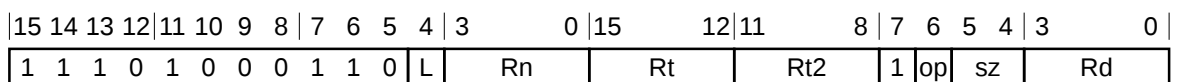
This section describes the encoding of the Load/store exclusive byte/half/dual instruction class. The encodings in this section are decoded from [Load/store exclusive](#), [load-acquire/store-release](#), [table branch](#).



| Decode fields | Instruction page |              |
|---------------|------------------|--------------|
| <b>L</b>      | <b>sz</b>        |              |
| 0             | 00               | STREXB       |
| 0             | 01               | STREXH       |
| 0             | 10               | Unallocated. |
| 0             | 11               | Unallocated. |
| 1             | 00               | LDREXB       |
| 1             | 01               | LDREXH       |
| 1             | 10               | Unallocated. |
| 1             | 11               | Unallocated. |

### Load-acquire/ Store-release

This section describes the encoding of the Load-acquire / Store-release instruction class. The encodings in this section are decoded from [Load/store exclusive](#), [load-acquire/store-release](#), [table branch](#).



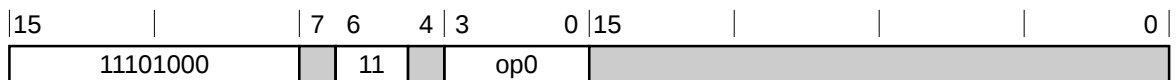
| Decode fields | Instruction page |           |              |
|---------------|------------------|-----------|--------------|
| <b>L</b>      | <b>op</b>        | <b>sz</b> |              |
| 0             | 0                | 00        | STLB         |
| 0             | 0                | 01        | STLH         |
| 0             | 0                | 10        | STL          |
| 0             | 0                | 11        | Unallocated. |
| 0             | 1                | 00        | STLEXB       |
| 0             | 1                | 01        | STLEXH       |
| 0             | 1                | 10        | STLEX        |
| 0             | 1                | 11        | Unallocated. |
| 1             | 0                | 00        | LDAB         |
| 1             | 0                | 01        | LDAH         |
| 1             | 0                | 10        | LDA          |
| 1             | 0                | 11        | Unallocated. |
| 1             | 1                | 00        | LDAEXB       |



| Decode fields |    |    | Instruction page |
|---------------|----|----|------------------|
| L             | op | sz |                  |
| 1             | 1  | 01 | LDAEXH           |
| 1             | 1  | 10 | LDAEX            |
| 1             | 1  | 11 | Unallocated.     |

### Load/store dual (post-indexed)

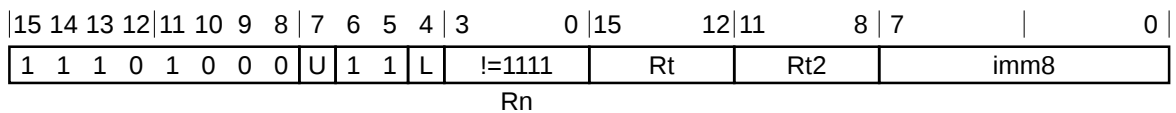
This section describes the encoding of the Load/store dual (post-indexed) group. The encodings in this section are decoded from [Load/store \(multiple, dual, exclusive, acquire-release\), table branch.unnumbered](#).



| Decode fields | Decode group or instruction page          |
|---------------|-------------------------------------------|
| op0           |                                           |
| 1111          | UNPREDICTABLE                             |
| != 1111       | Load/store dual (immediate, post-indexed) |

### Load/store dual (immediate, post-indexed)

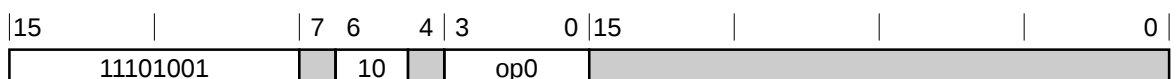
This section describes the encoding of the Load/store dual (immediate, post-indexed) instruction class. The encodings in this section are decoded from [Load/store dual \(post-indexed\)](#).



| Decode fields | Instruction page |
|---------------|------------------|
| L             |                  |
| 0             | STRD (immediate) |
| 1             | LDRD (immediate) |

### Load/store dual (literal and immediate)

This section describes the encoding of the Load/store dual (literal and immediate) group. The encodings in this section are decoded from [Load/store \(multiple, dual, exclusive, acquire-release\), table branch](#).

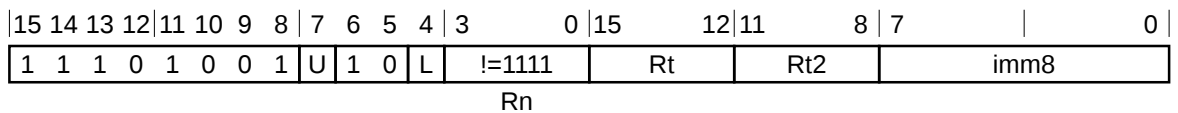


| Decode fields | Decode group or instruction page |
|---------------|----------------------------------|
| op0           |                                  |
| 1111          | LDRD (literal)                   |

| Decode fields | Decode group or instruction page |
|---------------|----------------------------------|
| op0           |                                  |
| != 1111       | Load/store dual (immediate)      |

### Load/store dual (immediate)

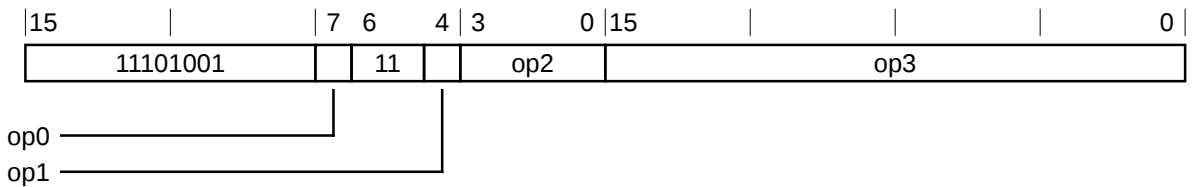
This section describes the encoding of the Load/store dual (immediate) instruction class. The encodings in this section are decoded from [Load/store dual \(literal and immediate\)](#).



| Decode fields | Instruction page |
|---------------|------------------|
| L             |                  |
| 0             | STRD (immediate) |
| 1             | LDRD (immediate) |

### Load/store dual (pre-indexed), secure gateway

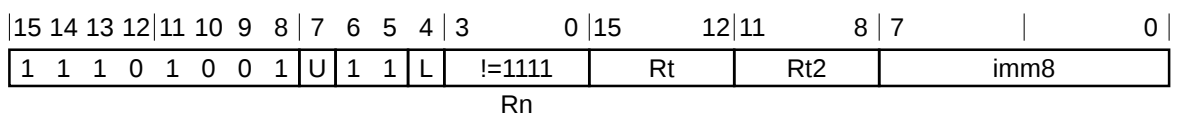
This section describes the encoding of the Load/store dual (pre-indexed), secure gateway group. The encodings in this section are decoded from [Load/store \(multiple, dual, exclusive, acquire-release\), table branch](#).



| Decode fields |     |         |                     | Decode group or instruction page         |
|---------------|-----|---------|---------------------|------------------------------------------|
| op0           | op1 | op2     | op3                 |                                          |
| 0             | 0   | 1111    | –                   | UNPREDICTABLE                            |
| 0             | 1   | 1111    | 1110100101111111    | SG                                       |
| 0             | 1   | 1111    | != 1110100101111111 | UNPREDICTABLE                            |
| 1             | 0   | 1111    | –                   | UNPREDICTABLE                            |
| 1             | 1   | 1111    | –                   | UNPREDICTABLE                            |
| –             | –   | != 1111 | –                   | Load/store dual (immediate, pre-indexed) |

### Load/store dual (immediate, pre-indexed)

This section describes the encoding of the Load/store dual (immediate, pre-indexed) instruction class. The encodings in this section are decoded from [Load/store dual \(pre-indexed\), secure gateway](#).



| Decode fields | Instruction page |
|---------------|------------------|
| L             |                  |
| 0             | STRD (immediate) |
| 1             | LDRD (immediate) |

### C2.3.2 Data-processing (shifted register)

This section describes the encoding of the Data-processing (shifted register) instruction class. The encodings in this section are decoded from [32-bit T32 instruction encoding](#).

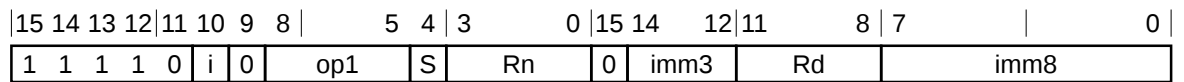
|    |    |    |    |    |    |   |     |   |    |     |      |    |      |      |    |   |   |   |   |   |   |   |
|----|----|----|----|----|----|---|-----|---|----|-----|------|----|------|------|----|---|---|---|---|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8   | 5 | 4  | 3   | 0    | 15 | 14   | 12   | 11 | 8 | 7 | 6 | 5 | 4 | 3 | 0 |
| 1  | 1  | 1  | 0  | 1  | 0  | 1 | op1 | S | Rn | (0) | imm3 | Rd | imm2 | type | Rm |   |   |   |   |   |   |   |

| Decode fields |   |         |                |         | Instruction page                                                |
|---------------|---|---------|----------------|---------|-----------------------------------------------------------------|
| op1           | S | Rn      | imm3:imm2:type | Rd      |                                                                 |
| 0000          | 0 | -       | -              | -       | AND (register) - AND, rotate right with extend variant          |
| 0000          | 1 | -       | != 0000011     | != 1111 | AND (register) - ANDS, shift or rotate by value variant         |
| 0000          | 1 | -       | != 0000011     | 1111    | TST (register) - Shift or rotate by value variant               |
| 0000          | 1 | -       | 0000011        | != 1111 | AND (register) - ANDS, rotate right with extend variant         |
| 0000          | 1 | -       | 0000011        | 1111    | TST (register) - Rotate right with extend variant               |
| 0001          | - | -       | -              | -       | BIC (register)                                                  |
| 0010          | 0 | != 1111 | -              | -       | ORR (register) - ORR, rotate right with extend variant          |
| 0010          | 0 | 1111    | -              | -       | MOV (register) - MOV, rotate right with extend variant          |
| 0010          | 1 | != 1111 | -              | -       | ORR (register) - ORRS, rotate right with extend variant         |
| 0010          | 1 | 1111    | -              | -       | MOV (register) - MOVS, rotate right with extend variant         |
| 0011          | 0 | != 1111 | -              | -       | ORN (register) - ORN, rotate right with extend variant          |
| 0011          | 0 | 1111    | -              | -       | MVN (register) - MVN, rotate right with extend variant          |
| 0011          | 1 | != 1111 | -              | -       | ORN (register) - ORNS, rotate right with extend variant         |
| 0011          | 1 | 1111    | -              | -       | MVN (register) - MVNS, rotate right with extend variant         |
| 0100          | 0 | -       | -              | -       | EOR (register) - EOR, rotate right with extend variant          |
| 0100          | 1 | -       | != 0000011     | != 1111 | EOR (register) - EORS, shift or rotate by value variant         |
| 0100          | 1 | -       | != 0000011     | 1111    | TEQ (register) - Shift or rotate by value variant               |
| 0100          | 1 | -       | 0000011        | != 1111 | EOR (register) - EORS, rotate right with extend variant         |
| 0100          | 1 | -       | 0000011        | 1111    | TEQ (register) - Rotate right with extend variant               |
| 0101          | - | -       | -              | -       | Unallocated.                                                    |
| 0110          | 0 | -       | xxxxx00        | -       | PKHBT, PKHTB - PKHBT variant                                    |
| 0110          | 0 | -       | xxxxx01        | -       | Unallocated.                                                    |
| 0110          | 0 | -       | xxxxx10        | -       | PKHBT, PKHTB - PKHTB variant                                    |
| 0110          | 0 | -       | xxxxx11        | -       | Unallocated.                                                    |
| 0111          | - | -       | -              | -       | Unallocated.                                                    |
| 1000          | 0 | != 1101 | -              | -       | ADD (register) - ADD, rotate right with extend variant          |
| 1000          | 0 | 1101    | -              | -       | ADD (SP plus register) - ADD, rotate right with extend variant  |
| 1000          | 1 | != 1101 | -              | != 1111 | ADD (register) - ADDS, rotate right with extend variant         |
| 1000          | 1 | 1101    | -              | != 1111 | ADD (SP plus register) - ADDS, rotate right with extend variant |
| 1000          | 1 | -       | -              | 1111    | CMN (register)                                                  |
| 1001          | - | -       | -              | -       | Unallocated.                                                    |
| 1010          | - | -       | -              | -       | ADC (register)                                                  |
| 1011          | - | -       | -              | -       | SBC (register)                                                  |
| 1100          | - | -       | -              | -       | Unallocated.                                                    |
| 1101          | 0 | != 1101 | -              | -       | SUB (register) - SUB, rotate right with extend variant          |

| Decode fields |   |         |                | Instruction page |                                                                                  |
|---------------|---|---------|----------------|------------------|----------------------------------------------------------------------------------|
| op1           | S | Rn      | imm3:imm2:type | Rd               |                                                                                  |
| 1101          | 0 | 1101    | -              | -                | <a href="#">SUB (SP minus register)</a> - SUB, rotate right with extend variant  |
| 1101          | 1 | != 1101 | -              | != 1111          | <a href="#">SUB (register)</a> - SUBS, rotate right with extend variant          |
| 1101          | 1 | 1101    | -              | != 1111          | <a href="#">SUB (SP minus register)</a> - SUBS, rotate right with extend variant |
| 1101          | 1 | -       | -              | 1111             | <a href="#">CMP (register)</a>                                                   |
| 1110          | - | -       | -              | -                | <a href="#">RSB (register)</a>                                                   |
| 1111          | - | -       | -              | -                | Unallocated.                                                                     |

### C2.3.3 Data-processing (modified immediate)

This section describes the encoding of the Data-processing (modified immediate) instruction class. The encodings in this section are decoded from [32-bit T32 instruction encoding](#).

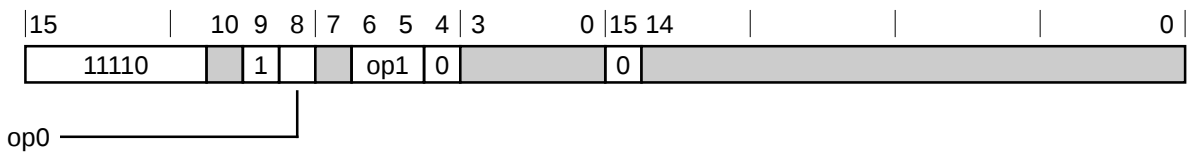


| Decode fields |   |         |         | Instruction page                                           |  |
|---------------|---|---------|---------|------------------------------------------------------------|--|
| op1           | S | Rn      | Rd      |                                                            |  |
| 0000          | 0 | -       | -       | <a href="#">AND (immediate)</a> - AND variant              |  |
| 0000          | 1 | -       | != 1111 | <a href="#">AND (immediate)</a> - ANDS variant             |  |
| 0000          | 1 | -       | 1111    | <a href="#">TST (immediate)</a>                            |  |
| 0001          | - | -       | -       | <a href="#">BIC (immediate)</a>                            |  |
| 0010          | 0 | != 1111 | -       | <a href="#">ORR (immediate)</a> - ORR variant              |  |
| 0010          | 0 | 1111    | -       | <a href="#">MOV (immediate)</a> - MOV variant              |  |
| 0010          | 1 | != 1111 | -       | <a href="#">ORR (immediate)</a> - ORRS variant             |  |
| 0010          | 1 | 1111    | -       | <a href="#">MOV (immediate)</a> - MOVS variant             |  |
| 0011          | 0 | != 1111 | -       | <a href="#">ORN (immediate)</a> - Non flag setting variant |  |
| 0011          | 0 | 1111    | -       | <a href="#">MVN (immediate)</a> - MVN variant              |  |
| 0011          | 1 | != 1111 | -       | <a href="#">ORN (immediate)</a> - Flag setting variant     |  |
| 0011          | 1 | 1111    | -       | <a href="#">MVN (immediate)</a> - MVNS variant             |  |
| 0100          | 0 | -       | -       | <a href="#">EOR (immediate)</a> - EOR variant              |  |
| 0100          | 1 | -       | != 1111 | <a href="#">EOR (immediate)</a> - EORS variant             |  |
| 0100          | 1 | -       | 1111    | <a href="#">TEQ (immediate)</a>                            |  |
| 0101          | - | -       | -       | Unallocated.                                               |  |
| 011x          | - | -       | -       | Unallocated.                                               |  |
| 1000          | 0 | != 1101 | -       | <a href="#">ADD (immediate)</a> - ADD variant              |  |
| 1000          | 0 | 1101    | -       | <a href="#">ADD (SP plus immediate)</a> - ADD variant      |  |
| 1000          | 1 | != 1101 | != 1111 | <a href="#">ADD (immediate)</a> - ADDS variant             |  |
| 1000          | 1 | 1101    | != 1111 | <a href="#">ADD (SP plus immediate)</a> - ADDS variant     |  |
| 1000          | 1 | -       | 1111    | <a href="#">CMN (immediate)</a>                            |  |
| 1001          | - | -       | -       | Unallocated.                                               |  |
| 1010          | - | -       | -       | <a href="#">ADC (immediate)</a>                            |  |
| 1011          | - | -       | -       | <a href="#">SBC (immediate)</a>                            |  |
| 1100          | - | -       | -       | Unallocated.                                               |  |
| 1101          | 0 | != 1101 | -       | <a href="#">SUB (immediate)</a> - SUB variant              |  |
| 1101          | 0 | 1101    | -       | <a href="#">SUB (SP minus immediate)</a> - SUB variant     |  |
| 1101          | 1 | != 1101 | != 1111 | <a href="#">SUB (immediate)</a> - SUBS variant             |  |
| 1101          | 1 | 1101    | != 1111 | <a href="#">SUB (SP minus immediate)</a> - SUBS variant    |  |
| 1101          | 1 | -       | 1111    | <a href="#">CMP (immediate)</a>                            |  |

| Decode fields |   |    |    | Instruction page |
|---------------|---|----|----|------------------|
| op1           | S | Rn | Rd |                  |
| 1110          | - | -  | -  | RSB (immediate)  |
| 1111          | - | -  | -  | Unallocated.     |

### C2.3.4 Data-processing (plain binary immediate)

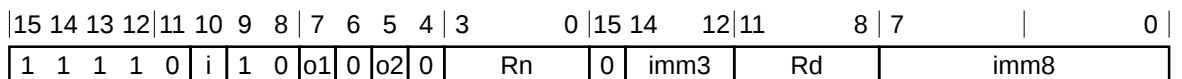
This section describes the encoding of the Data-processing (plain binary immediate) group. The encodings in this section are decoded from [32-bit T32 instruction encoding](#).



| Decode fields |     | Decode group or instruction page   |
|---------------|-----|------------------------------------|
| op0           | op1 |                                    |
| 0             | 0x  | Data-processing (simple immediate) |
| 0             | 10  | Move Wide (16-bit immediate)       |
| 0             | 11  | Unallocated.                       |
| 1             | -   | Saturate, Bitfield                 |

### Data-processing (simple immediate)

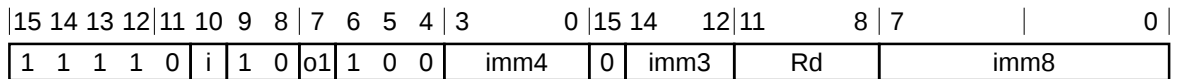
This section describes the encoding of the Data-processing (simple immediate) instruction class. The encodings in this section are decoded from [Data-processing \(plain binary immediate\)](#).



| Decode fields |    |         | Instruction page         |
|---------------|----|---------|--------------------------|
| o1            | o2 | Rn      |                          |
| 0             | 0  | != 11x1 | ADD (immediate)          |
| 0             | 0  | 1101    | ADD (SP plus immediate)  |
| 0             | 0  | 1111    | ADR - T3                 |
| 0             | 1  | -       | Unallocated.             |
| 1             | 0  | -       | Unallocated.             |
| 1             | 1  | != 11x1 | SUB (immediate)          |
| 1             | 1  | 1101    | SUB (SP minus immediate) |
| 1             | 1  | 1111    | ADR - T2                 |

### Move Wide (16-bit immediate)

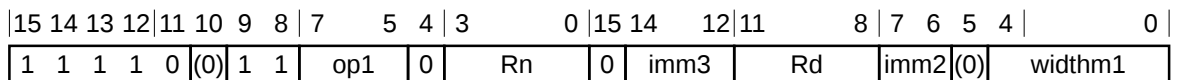
This section describes the encoding of the Move Wide (16-bit immediate) instruction class. The encodings in this section are decoded from [Data-processing \(plain binary immediate\)](#).



| Decode fields | Instruction page |
|---------------|------------------|
| <b>op1</b>    |                  |
| 0             | MOV (immediate)  |
| 1             | MOVT             |

### Saturate, Bitfield

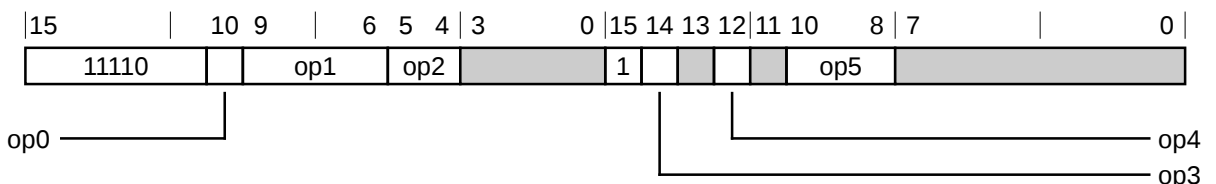
This section describes the encoding of the Saturate, Bitfield instruction class. The encodings in this section are decoded from [Data-processing \(plain binary immediate\)](#).



| Decode fields |         |           | Instruction page                      |
|---------------|---------|-----------|---------------------------------------|
| op1           | Rn      | imm3:imm2 |                                       |
| 000           | -       | -         | SSAT - Logical shift left variant     |
| 001           | -       | != 00000  | SSAT - Arithmetic shift right variant |
| 001           | -       | 00000     | SSAT16                                |
| 010           | -       | -         | SBFX                                  |
| 011           | != 1111 | -         | BFI                                   |
| 011           | 1111    | -         | BFC                                   |
| 100           | -       | -         | USAT - Logical shift left variant     |
| 101           | -       | != 00000  | USAT - Arithmetic shift right variant |
| 101           | -       | 00000     | USAT16                                |
| 110           | -       | -         | UBFX                                  |
| 111           | -       | -         | Unallocated.                          |

### C2.3.5 Branches and miscellaneous control

This section describes the encoding of the Branches and miscellaneous control group. The encodings in this section are decoded from [32-bit T32 instruction encoding](#).



| Decode fields |      |     |     |     |     | Decode group or instruction page |
|---------------|------|-----|-----|-----|-----|----------------------------------|
| op0           | op1  | op2 | op3 | op4 | op5 |                                  |
| 0             | 1110 | 0x  | 0   | 0   | -   | MSR (register)                   |
| 0             | 1110 | 10  | 0   | 0   | 000 | Hints                            |

| Decode fields |         |     |     |     |        | Decode group or instruction page |
|---------------|---------|-----|-----|-----|--------|----------------------------------|
| op0           | op1     | op2 | op3 | op4 | op5    |                                  |
| 0             | 1110    | 10  | 0   | 0   | != 000 | Unallocated.                     |
| 0             | 1110    | 11  | 0   | 0   | -      | Miscellaneous system             |
| 0             | 1111    | 0x  | 0   | 0   | -      | Unallocated.                     |
| 0             | 1111    | 1x  | 0   | 0   | -      | MRS                              |
| 1             | 1110    | -   | 0   | 0   | -      | Unallocated.                     |
| 1             | 1111    | 0x  | 0   | 0   | -      | Unallocated.                     |
| 1             | 1111    | 1x  | 0   | 0   | -      | Exception generation             |
| -             | != 111x | -   | 0   | 0   | -      | B - T3 variant                   |
| -             | -       | -   | 0   | 1   | -      | B - T4 variant                   |
| -             | -       | -   | 1   | 0   | -      | Unallocated.                     |
| -             | -       | -   | 1   | 1   | -      | BL                               |

## Hints

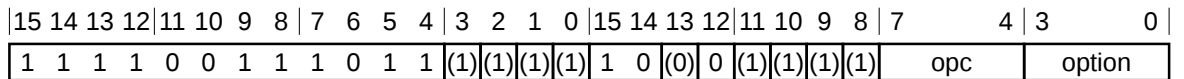
This section describes the encoding of the Hints instruction class. The encodings in this section are decoded from [Branches and miscellaneous control](#).

|    |    |    |    |    |    |   |   |   |   |   |   |     |     |     |     |    |    |     |    |     |    |   |   |      |        |   |   |
|----|----|----|----|----|----|---|---|---|---|---|---|-----|-----|-----|-----|----|----|-----|----|-----|----|---|---|------|--------|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3   | 2   | 1   | 0   | 15 | 14 | 13  | 12 | 11  | 10 | 9 | 8 | 7    | 4      | 3 | 0 |
| 1  | 1  | 1  | 1  | 0  | 0  | 1 | 1 | 1 | 0 | 1 | 0 | (1) | (1) | (1) | (1) | 1  | 0  | (0) | 0  | (0) | 0  | 0 | 0 | hint | option |   |   |

| Decode fields |        | Instruction page               |
|---------------|--------|--------------------------------|
| hint          | option |                                |
| 0000          | 0000   | NOP                            |
| 0000          | 0001   | YIELD                          |
| 0000          | 0010   | WFE                            |
| 0000          | 0011   | WFI                            |
| 0000          | 0100   | SEV                            |
| 0000          | 0101   | Reserved hint, behaves as NOP. |
| 0000          | 011x   | Reserved hint, behaves as NOP. |
| 0000          | 1xxx   | Reserved hint, behaves as NOP. |
| 0001          | !=0100 | Reserved hint, behaves as NOP. |
| 0001          | 0100   | CSDB                           |
| 0001          | 1xxx   | Reserved hint, behaves as NOP. |
| 001x          | -      | Reserved hint, behaves as NOP. |
| 01xx          | -      | Reserved hint, behaves as NOP. |
| 10xx          | -      | Reserved hint, behaves as NOP. |
| 110x          | -      | Reserved hint, behaves as NOP. |
| 1110          | -      | Reserved hint, behaves as NOP. |
| 1111          | -      | DBG                            |

## Miscellaneous system

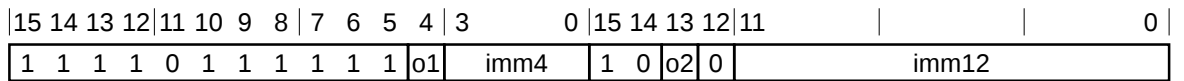
This section describes the encoding of the Miscellaneous system instruction class. The encodings in this section are decoded from [Branches and miscellaneous control](#).



| Decode fields<br>opc | option | Instruction page      |
|----------------------|--------|-----------------------|
| 000x                 |        | Unallocated.          |
| 0010                 |        | <a href="#">CLREX</a> |
| 0011                 |        | Unallocated.          |
| 0100                 | 1111   | <a href="#">DSB</a>   |
| 0100                 | 0000   | <a href="#">SSBB</a>  |
| 0100                 | 0100   | <a href="#">PSSBB</a> |
| 0101                 |        | <a href="#">DMB</a>   |
| 0110                 |        | <a href="#">ISB</a>   |
| 0111                 |        | Unallocated.          |
| 1xxx                 |        | Unallocated.          |

### Exception generation

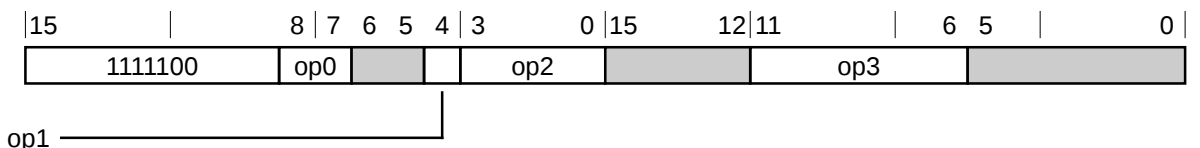
This section describes the encoding of the Exception generation instruction class. The encodings in this section are decoded from [Branches and miscellaneous control](#).



| Decode fields<br>o1 | o2 | Instruction page    |
|---------------------|----|---------------------|
| 0                   | 0  | Unallocated.        |
| 0                   | 1  | Unallocated.        |
| 1                   | 0  | Unallocated.        |
| 1                   | 1  | <a href="#">UDF</a> |

### C2.3.6 Load/store single

This section describes the encoding of the Load/store single group. The encodings in this section are decoded from [32-bit T32 instruction encoding](#).



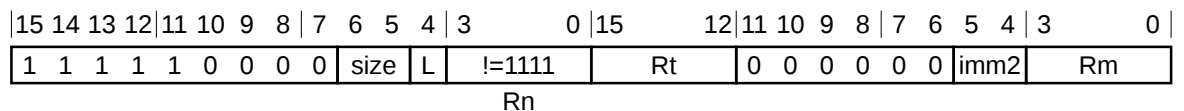
| Decode fields |     |         |        | Decode group or instruction page                       |  |
|---------------|-----|---------|--------|--------------------------------------------------------|--|
| op0           | op1 | op2     | op3    |                                                        |  |
| 00            | -   | != 1111 | 000000 | <a href="#">Load/store, unsigned (register offset)</a> |  |
| 00            | -   | != 1111 | 000001 | Unallocated.                                           |  |



| Decode fields |     |         |        | op3 | Decode group or instruction page               |
|---------------|-----|---------|--------|-----|------------------------------------------------|
| op0           | op1 | op2     |        |     |                                                |
| 00            | -   | != 1111 | 00001x |     | Unallocated.                                   |
| 00            | -   | != 1111 | 0001xx |     | Unallocated.                                   |
| 00            | -   | != 1111 | 001xxx |     | Unallocated.                                   |
| 00            | -   | != 1111 | 01xxxx |     | Unallocated.                                   |
| 00            | -   | != 1111 | 10x0xx |     | Unallocated.                                   |
| 00            | -   | != 1111 | 10x1xx |     | Load/store, unsigned (immediate, post-indexed) |
| 00            | -   | != 1111 | 1100xx |     | Load/store, unsigned (negative immediate)      |
| 00            | -   | != 1111 | 1110xx |     | Load/store, unsigned (unprivileged)            |
| 00            | -   | != 1111 | 11x1xx |     | Load/store, unsigned (immediate, pre-indexed)  |
| 01            | -   | != 1111 | -      |     | Load/store, unsigned (positive immediate)      |
| 0x            | -   | 1111    | -      |     | Load, unsigned (literal)                       |
| 10            | 1   | != 1111 | 000000 |     | Load/store, signed (register offset)           |
| 10            | 1   | != 1111 | 000001 |     | Unallocated.                                   |
| 10            | 1   | != 1111 | 00001x |     | Unallocated.                                   |
| 10            | 1   | != 1111 | 0001xx |     | Unallocated.                                   |
| 10            | 1   | != 1111 | 001xxx |     | Unallocated.                                   |
| 10            | 1   | != 1111 | 01xxxx |     | Unallocated.                                   |
| 10            | 1   | != 1111 | 10x0xx |     | Unallocated.                                   |
| 10            | 1   | != 1111 | 10x1xx |     | Load/store, signed (immediate, post-indexed)   |
| 10            | 1   | != 1111 | 1100xx |     | Load/store, signed (negative immediate)        |
| 10            | 1   | != 1111 | 1110xx |     | Load/store, signed (unprivileged)              |
| 10            | 1   | != 1111 | 11x1xx |     | Load/store, signed (immediate, pre-indexed)    |
| 11            | 1   | != 1111 | -      |     | Load/store, signed (positive immediate)        |
| 1x            | 1   | 1111    | -      |     | Load, signed (literal)                         |

### Load/store, unsigned (register offset)

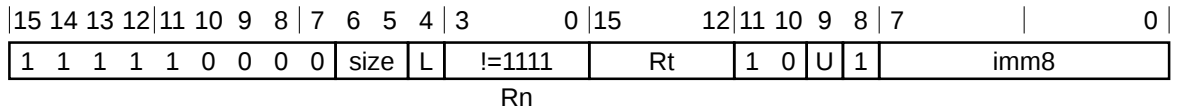
This section describes the encoding of the Load/store, unsigned (register offset) instruction class. The encodings in this section are decoded from [Load/store single](#).



| Decode fields |   |         | Instruction page                             |
|---------------|---|---------|----------------------------------------------|
| size          | L | Rt      |                                              |
| 00            | 0 | -       | STRB (register)                              |
| 00            | 1 | != 1111 | LDRB (register)                              |
| 00            | 1 | 1111    | PLD, PLDW (register) - Preload read variant  |
| 01            | 0 | -       | STRH (register)                              |
| 01            | 1 | != 1111 | LDRH (register)                              |
| 01            | 1 | 1111    | PLD, PLDW (register) - Preload write variant |
| 10            | 0 | -       | STR (register)                               |
| 10            | 1 | -       | LDR (register)                               |
| 11            | - | -       | Unallocated.                                 |

### Load/store, unsigned (immediate, post-indexed)

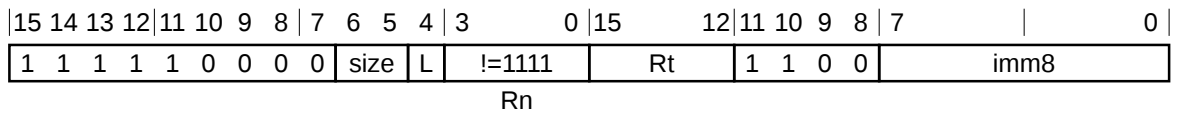
This section describes the encoding of the Load/store, unsigned (immediate, post-indexed) instruction class. The encodings in this section are decoded from [Load/store single](#).



| Decode fields |   | Instruction page |
|---------------|---|------------------|
| size          | L |                  |
| 00            | 0 | STRB (immediate) |
| 00            | 1 | LDRB (immediate) |
| 01            | 0 | STRH (immediate) |
| 01            | 1 | LDRH (immediate) |
| 10            | 0 | STR (immediate)  |
| 10            | 1 | LDR (immediate)  |
| 11            | - | Unallocated.     |

### Load/store, unsigned (negative immediate)

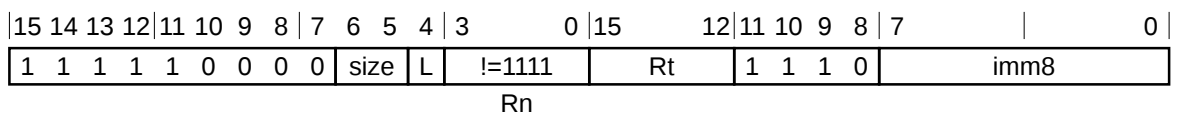
This section describes the encoding of the Load/store, unsigned (negative immediate) instruction class. The encodings in this section are decoded from [Load/store single](#).



| Decode fields |   |         | Instruction page                              |
|---------------|---|---------|-----------------------------------------------|
| size          | L | Rt      |                                               |
| 00            | 0 | -       | STRB (immediate)                              |
| 00            | 1 | != 1111 | LDRB (immediate)                              |
| 00            | 1 | 1111    | PLD, PLDW (immediate) - Preload read variant  |
| 01            | 0 | -       | STRH (immediate)                              |
| 01            | 1 | != 1111 | LDRH (immediate)                              |
| 01            | 1 | 1111    | PLD, PLDW (immediate) - Preload write variant |
| 10            | 0 | -       | STR (immediate)                               |
| 10            | 1 | -       | LDR (immediate)                               |
| 11            | - | -       | Unallocated.                                  |

### Load/store, unsigned (unprivileged)

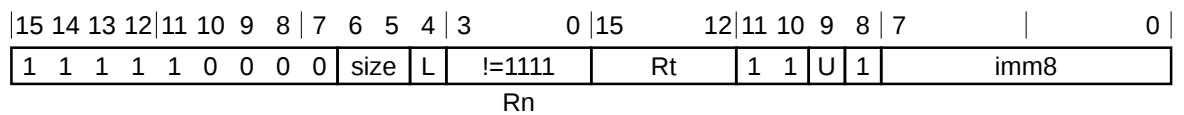
This section describes the encoding of the Load/store, unsigned (unprivileged) instruction class. The encodings in this section are decoded from [Load/store single](#).



| Decode fields |   | Instruction page |
|---------------|---|------------------|
| size          | L |                  |
| 00            | 0 | STRBT            |
| 00            | 1 | LDRBT            |
| 01            | 0 | STRHT            |
| 01            | 1 | LDRHT            |
| 10            | 0 | STRT             |
| 10            | 1 | LDRT             |
| 11            | – | Unallocated.     |

### Load/store, unsigned (immediate, pre-indexed)

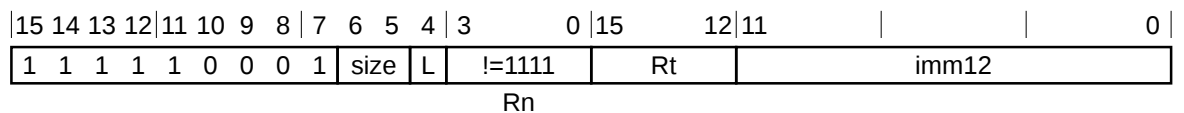
This section describes the encoding of the Load/store, unsigned (immediate, pre-indexed) instruction class. The encodings in this section are decoded from [Load/store single](#).



| Decode fields |   | Instruction page |
|---------------|---|------------------|
| size          | L |                  |
| 00            | 0 | STRB (immediate) |
| 00            | 1 | LDRB (immediate) |
| 01            | 0 | STRH (immediate) |
| 01            | 1 | LDRH (immediate) |
| 10            | 0 | STR (immediate)  |
| 10            | 1 | LDR (immediate)  |
| 11            | – | Unallocated.     |

### Load/store, unsigned (positive immediate)

This section describes the encoding of the Load/store, unsigned (positive immediate) instruction class. The encodings in this section are decoded from [Load/store single](#).



| Decode fields |   |         | Instruction page                              |
|---------------|---|---------|-----------------------------------------------|
| size          | L | Rt      |                                               |
| 00            | 0 | –       | STRB (immediate)                              |
| 00            | 1 | != 1111 | LDRB (immediate)                              |
| 00            | 1 | 1111    | PLD, PLDW (immediate) - Preload read variant  |
| 01            | 0 | –       | STRH (immediate)                              |
| 01            | 1 | != 1111 | LDRH (immediate)                              |
| 01            | 1 | 1111    | PLD, PLDW (immediate) - Preload write variant |
| 10            | 0 | –       | STR (immediate)                               |
| 10            | 1 | –       | LDR (immediate)                               |



| Decode fields size | Instruction page |
|--------------------|------------------|
| 1x                 | Unallocated.     |

### Load/store, signed (negative immediate)

This section describes the encoding of the Load/store, signed (negative immediate) instruction class. The encodings in this section are decoded from [Load/store single](#).

|    |    |    |    |    |    |   |   |   |      |   |        |    |   |    |    |    |      |   |   |   |   |
|----|----|----|----|----|----|---|---|---|------|---|--------|----|---|----|----|----|------|---|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6    | 5 | 4      | 3  | 0 | 15 | 12 | 11 | 10   | 9 | 8 | 7 | 0 |
| 1  | 1  | 1  | 1  | 1  | 0  | 0 | 1 | 0 | size | 1 | !=1111 | Rt | 1 | 1  | 0  | 0  | imm8 |   |   |   |   |

Rn

| Decode fields size | Rt      | Instruction page                                |
|--------------------|---------|-------------------------------------------------|
| 00                 | –       | <a href="#">LDRSB (immediate)</a>               |
| 00                 | 1111    | <a href="#">PLI (immediate, literal)</a>        |
| 01                 | != 1111 | <a href="#">LDRSH (immediate)</a>               |
| 01                 | 1111    | Reserved hint, behaves as <a href="#">NOP</a> . |
| 1x                 | –       | Unallocated.                                    |

### Load/store, signed (unprivileged)

This section describes the encoding of the Load/store, signed (unprivileged) instruction class. The encodings in this section are decoded from [Load/store single](#).

|    |    |    |    |    |    |   |   |   |      |   |        |    |   |    |    |    |      |   |   |   |   |
|----|----|----|----|----|----|---|---|---|------|---|--------|----|---|----|----|----|------|---|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6    | 5 | 4      | 3  | 0 | 15 | 12 | 11 | 10   | 9 | 8 | 7 | 0 |
| 1  | 1  | 1  | 1  | 1  | 0  | 0 | 1 | 0 | size | 1 | !=1111 | Rt | 1 | 1  | 1  | 0  | imm8 |   |   |   |   |

Rn

| Decode fields size | Instruction page       |
|--------------------|------------------------|
| 00                 | <a href="#">LDRSBT</a> |
| 01                 | <a href="#">LDRSHT</a> |
| 1x                 | Unallocated.           |

### Load/store, signed (immediate, pre-indexed)

This section describes the encoding of the Load/store, signed (immediate, pre-indexed) instruction class. The encodings in this section are decoded from [Load/store single](#).

|    |    |    |    |    |    |   |   |   |      |   |        |    |   |    |    |    |      |   |   |   |   |
|----|----|----|----|----|----|---|---|---|------|---|--------|----|---|----|----|----|------|---|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6    | 5 | 4      | 3  | 0 | 15 | 12 | 11 | 10   | 9 | 8 | 7 | 0 |
| 1  | 1  | 1  | 1  | 1  | 0  | 0 | 1 | 0 | size | 1 | !=1111 | Rt | 1 | 1  | U  | 1  | imm8 |   |   |   |   |

Rn

| Decode fields size | Instruction page  |
|--------------------|-------------------|
| 00                 | LDRSB (immediate) |
| 01                 | LDRSH (immediate) |
| 1x                 | Unallocated.      |

### Load/store, signed (positive immediate)

This section describes the encoding of the Load/store, signed (positive immediate) instruction class. The encodings in this section are decoded from [Load/store single](#).

|    |    |    |    |    |    |   |   |   |      |   |   |       |    |    |    |    |  |  |       |
|----|----|----|----|----|----|---|---|---|------|---|---|-------|----|----|----|----|--|--|-------|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6    | 5 | 4 | 3     | 0  | 15 | 12 | 11 |  |  | 0     |
| 1  | 1  | 1  | 1  | 1  | 0  | 0 | 1 | 1 | size | 1 | ! | =1111 | Rt |    |    |    |  |  | imm12 |

Rn

| Decode fields size | Rt      | Instruction page                       |
|--------------------|---------|----------------------------------------|
| 00                 | != 1111 | LDRSB (immediate)                      |
| 00                 | 1111    | PLI (immediate, literal)               |
| 01                 | != 1111 | LDRSH (immediate)                      |
| 01                 | 1111    | Reserved hint, behaves as <b>NOP</b> . |

### Load, signed (literal)

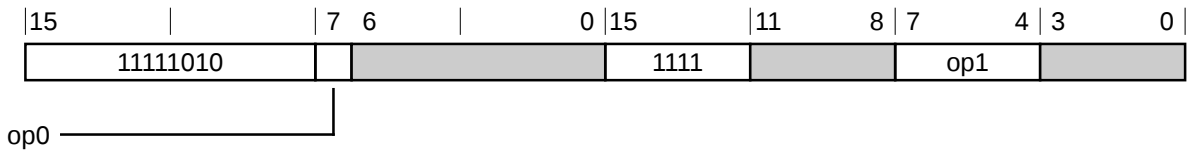
This section describes the encoding of the Load, signed (literal) instruction class. The encodings in this section are decoded from [Load/store single](#).

|    |    |    |    |    |    |   |   |   |      |   |   |   |   |   |   |    |    |    |  |  |       |
|----|----|----|----|----|----|---|---|---|------|---|---|---|---|---|---|----|----|----|--|--|-------|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6    | 5 | 4 | 3 | 2 | 1 | 0 | 15 | 12 | 11 |  |  | 0     |
| 1  | 1  | 1  | 1  | 1  | 0  | 0 | 1 | U | size | 1 | 1 | 1 | 1 | 1 | 1 | Rt |    |    |  |  | imm12 |

| Decode fields size | Rt      | Instruction page                       |
|--------------------|---------|----------------------------------------|
| 00                 | != 1111 | LDRSB (literal)                        |
| 00                 | 1111    | PLI (immediate, literal)               |
| 01                 | != 1111 | LDRSH (literal)                        |
| 01                 | 1111    | Reserved hint, behaves as <b>NOP</b> . |
| 1x                 | -       | Unallocated.                           |

### C2.3.7 Data-processing (register)

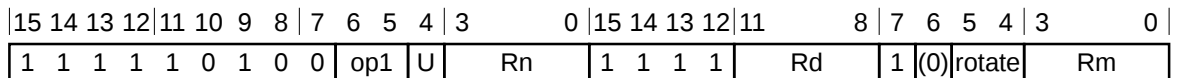
This section describes the encoding of the Data-processing (register) group. The encodings in this section are decoded from [32-bit T32 instruction encoding](#).



| Decode fields |      | Decode group or instruction page                             |
|---------------|------|--------------------------------------------------------------|
| op0           | op1  |                                                              |
| 0             | 0000 | MOV, MOVS (register-shifted register) - Flag setting variant |
| 0             | 0001 | Unallocated.                                                 |
| 0             | 001x | Unallocated.                                                 |
| 0             | 01xx | Unallocated.                                                 |
| 0             | 1xxx | Register extends                                             |
| 1             | 0xxx | Parallel add-subtract                                        |
| 1             | 10xx | Data-processing (two source registers)                       |
| 1             | 11xx | Unallocated.                                                 |

### C2.3.7.1 Register extends

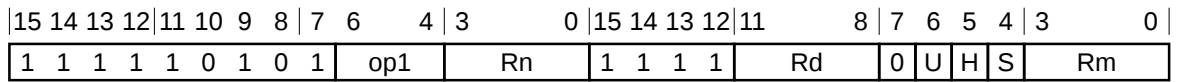
This section describes the encoding of the Register extends instruction class. The encodings in this section are decoded from [Data-processing \(register\)](#).



| Decode fields |   |         | Instruction page |
|---------------|---|---------|------------------|
| op1           | U | Rn      |                  |
| 00            | 0 | != 1111 | SXTAH            |
| 00            | 0 | 1111    | SXTH             |
| 00            | 1 | != 1111 | UXTAH            |
| 00            | 1 | 1111    | UXTH             |
| 01            | 0 | != 1111 | SXTAB16          |
| 01            | 0 | 1111    | SXTB16           |
| 01            | 1 | != 1111 | UXTAB16          |
| 01            | 1 | 1111    | UXTB16           |
| 10            | 0 | != 1111 | SXTAB            |
| 10            | 0 | 1111    | SXTB             |
| 10            | 1 | != 1111 | UXTAB            |
| 10            | 1 | 1111    | UXTB             |
| 11            | - | -       | Unallocated.     |

### Parallel add-subtract

This section describes the encoding of the Parallel add-subtract instruction class. The encodings in this section are decoded from [Data-processing \(register\)](#).



| Decode fields | Instruction page |   |   |                         |
|---------------|------------------|---|---|-------------------------|
| op1           | U                | H | S |                         |
| 000           | 0                | 0 | 0 | <a href="#">SADD8</a>   |
| 000           | 0                | 0 | 1 | <a href="#">QADD8</a>   |
| 000           | 0                | 1 | 0 | <a href="#">SHADD8</a>  |
| 000           | 0                | 1 | 1 | Unallocated.            |
| 000           | 1                | 0 | 0 | <a href="#">UADD8</a>   |
| 000           | 1                | 0 | 1 | <a href="#">UQADD8</a>  |
| 000           | 1                | 1 | 0 | <a href="#">UHADD8</a>  |
| 000           | 1                | 1 | 1 | Unallocated.            |
| 001           | 0                | 0 | 0 | <a href="#">SADD16</a>  |
| 001           | 0                | 0 | 1 | <a href="#">QADD16</a>  |
| 001           | 0                | 1 | 0 | <a href="#">SHADD16</a> |
| 001           | 0                | 1 | 1 | Unallocated.            |
| 001           | 1                | 0 | 0 | <a href="#">UADD16</a>  |
| 001           | 1                | 0 | 1 | <a href="#">UQADD16</a> |
| 001           | 1                | 1 | 0 | <a href="#">UHADD16</a> |
| 001           | 1                | 1 | 1 | Unallocated.            |
| 010           | 0                | 0 | 0 | <a href="#">SASX</a>    |
| 010           | 0                | 0 | 1 | <a href="#">QASX</a>    |
| 010           | 0                | 1 | 0 | <a href="#">SHASX</a>   |
| 010           | 0                | 1 | 1 | Unallocated.            |
| 010           | 1                | 0 | 0 | <a href="#">UASX</a>    |
| 010           | 1                | 0 | 1 | <a href="#">UQASX</a>   |
| 010           | 1                | 1 | 0 | <a href="#">UHASX</a>   |
| 010           | 1                | 1 | 1 | Unallocated.            |
| 100           | 0                | 0 | 0 | <a href="#">SSUB8</a>   |
| 100           | 0                | 0 | 1 | <a href="#">QSUB8</a>   |
| 100           | 0                | 1 | 0 | <a href="#">SHSUB8</a>  |
| 100           | 0                | 1 | 1 | Unallocated.            |
| 100           | 1                | 0 | 0 | <a href="#">USUB8</a>   |
| 100           | 1                | 0 | 1 | <a href="#">UQSUB8</a>  |
| 100           | 1                | 1 | 0 | <a href="#">UHSUB8</a>  |
| 100           | 1                | 1 | 1 | Unallocated.            |
| 101           | 0                | 0 | 0 | <a href="#">SSUB16</a>  |
| 101           | 0                | 0 | 1 | <a href="#">QSUB16</a>  |
| 101           | 0                | 1 | 0 | <a href="#">SHSUB16</a> |
| 101           | 0                | 1 | 1 | Unallocated.            |
| 101           | 1                | 0 | 0 | <a href="#">USUB16</a>  |
| 101           | 1                | 0 | 1 | <a href="#">UQSUB16</a> |
| 101           | 1                | 1 | 0 | <a href="#">UHSUB16</a> |
| 101           | 1                | 1 | 1 | Unallocated.            |
| 110           | 0                | 0 | 0 | <a href="#">SSAX</a>    |
| 110           | 0                | 0 | 1 | <a href="#">QSAX</a>    |
| 110           | 0                | 1 | 0 | <a href="#">SHSAX</a>   |
| 110           | 0                | 1 | 1 | Unallocated.            |
| 110           | 1                | 0 | 0 | <a href="#">USAX</a>    |
| 110           | 1                | 0 | 1 | <a href="#">UQSAX</a>   |
| 110           | 1                | 1 | 0 | <a href="#">UHSAX</a>   |



| Decode fields<br>op1 | Decode fields |   |   | Instruction page |
|----------------------|---------------|---|---|------------------|
|                      | U             | H | S |                  |
| 110                  | 1             | 1 | 1 | Unallocated.     |
| 111                  | -             | - | - | Unallocated.     |

### Data-processing (two source registers)

This section describes the encoding of the Data-processing (two source registers) instruction class. The encodings in this section are decoded from [Data-processing \(register\)](#).

|    |    |    |    |    |    |   |   |   |     |   |   |    |    |    |    |    |    |    |   |   |   |     |   |    |  |
|----|----|----|----|----|----|---|---|---|-----|---|---|----|----|----|----|----|----|----|---|---|---|-----|---|----|--|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6   | 4 | 3 | 0  | 15 | 14 | 13 | 12 | 11 | 8  | 7 | 6 | 5 | 4   | 3 | 0  |  |
| 1  | 1  | 1  | 1  | 1  | 0  | 1 | 0 | 1 | op1 |   |   | Rn |    | 1  | 1  | 1  | 1  | Rd |   | 1 | 0 | op2 |   | Rm |  |

| Decode fields<br>op1 | Decode fields<br>op2 | Instruction page      |
|----------------------|----------------------|-----------------------|
|                      |                      |                       |
| 000                  | 01                   | <a href="#">QDADD</a> |
| 000                  | 10                   | <a href="#">QSUB</a>  |
| 000                  | 11                   | <a href="#">QDSUB</a> |
| 001                  | 00                   | <a href="#">REV</a>   |
| 001                  | 01                   | <a href="#">REV16</a> |
| 001                  | 10                   | <a href="#">RBIT</a>  |
| 001                  | 11                   | <a href="#">REVSH</a> |
| 010                  | 00                   | <a href="#">SEL</a>   |
| 010                  | 01                   | Unallocated.          |
| 010                  | 1x                   | Unallocated.          |
| 011                  | 00                   | <a href="#">CLZ</a>   |
| 011                  | 01                   | Unallocated.          |
| 011                  | 1x                   | Unallocated.          |
| 1xx                  | -                    | Unallocated.          |

### C2.3.8 Multiply, multiply accumulate, and absolute difference

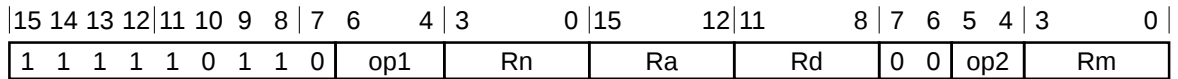
This section describes the encoding of the Multiply, multiply accumulate, and absolute difference group. The encodings in this section are decoded from [32-bit T32 instruction encoding](#).

|           |   |   |    |   |   |     |   |   |
|-----------|---|---|----|---|---|-----|---|---|
| 15        | 6 | 0 | 15 | 8 | 7 | 6   | 5 | 0 |
| 111110110 |   |   |    |   |   | op0 |   |   |

| Decode fields<br>op0 | Decode group or instruction page                 |
|----------------------|--------------------------------------------------|
| 00                   | <a href="#">Multiply and absolute difference</a> |
| 01                   | Unallocated.                                     |
| 1x                   | Unallocated.                                     |

## Multiply and absolute difference

This section describes the encoding of the Multiply and absolute difference instruction class. The encodings in this section are decoded from [Multiply](#), [multiply accumulate](#), and [absolute difference](#).



| Decode fields |         |     | Instruction page                                                                                                                   |
|---------------|---------|-----|------------------------------------------------------------------------------------------------------------------------------------|
| op1           | Ra      | op2 |                                                                                                                                    |
| 000           | != 1111 | 00  | <a href="#">MLA</a>                                                                                                                |
| 000           | -       | 01  | <a href="#">MLS</a>                                                                                                                |
| 000           | -       | 1x  | Unallocated.                                                                                                                       |
| 000           | 1111    | 00  | <a href="#">MUL</a>                                                                                                                |
| 001           | != 1111 | 00  | <a href="#">SMLABB</a> , <a href="#">SMLABT</a> , <a href="#">SMLATB</a> , <a href="#">SMLATT</a> - <a href="#">SMLABB</a> variant |
| 001           | != 1111 | 01  | <a href="#">SMLABB</a> , <a href="#">SMLABT</a> , <a href="#">SMLATB</a> , <a href="#">SMLATT</a> - <a href="#">SMLABT</a> variant |
| 001           | != 1111 | 10  | <a href="#">SMLABB</a> , <a href="#">SMLABT</a> , <a href="#">SMLATB</a> , <a href="#">SMLATT</a> - <a href="#">SMLATB</a> variant |
| 001           | != 1111 | 11  | <a href="#">SMLABB</a> , <a href="#">SMLABT</a> , <a href="#">SMLATB</a> , <a href="#">SMLATT</a> - <a href="#">SMLATT</a> variant |
| 001           | 1111    | 00  | <a href="#">SMULBB</a> , <a href="#">SMULBT</a> , <a href="#">SMULTB</a> , <a href="#">SMULTT</a> - <a href="#">SMULBB</a> variant |
| 001           | 1111    | 01  | <a href="#">SMULBB</a> , <a href="#">SMULBT</a> , <a href="#">SMULTB</a> , <a href="#">SMULTT</a> - <a href="#">SMULBT</a> variant |
| 001           | 1111    | 10  | <a href="#">SMULBB</a> , <a href="#">SMULBT</a> , <a href="#">SMULTB</a> , <a href="#">SMULTT</a> - <a href="#">SMULTB</a> variant |
| 001           | 1111    | 11  | <a href="#">SMULBB</a> , <a href="#">SMULBT</a> , <a href="#">SMULTB</a> , <a href="#">SMULTT</a> - <a href="#">SMULTT</a> variant |
| 010           | != 1111 | 00  | <a href="#">SMLAD</a> , <a href="#">SMLADX</a> - <a href="#">SMLAD</a> variant                                                     |
| 010           | != 1111 | 01  | <a href="#">SMLAD</a> , <a href="#">SMLADX</a> - <a href="#">SMLADX</a> variant                                                    |
| 010           | -       | 1x  | Unallocated.                                                                                                                       |
| 010           | 1111    | 00  | <a href="#">SMUAD</a> , <a href="#">SMUADX</a> - <a href="#">SMUAD</a> variant                                                     |
| 010           | 1111    | 01  | <a href="#">SMUAD</a> , <a href="#">SMUADX</a> - <a href="#">SMUADX</a> variant                                                    |
| 011           | != 1111 | 00  | <a href="#">SMLAWB</a> , <a href="#">SMLAWT</a> - <a href="#">SMLAWB</a> variant                                                   |
| 011           | != 1111 | 01  | <a href="#">SMLAWB</a> , <a href="#">SMLAWT</a> - <a href="#">SMLAWT</a> variant                                                   |
| 011           | -       | 1x  | Unallocated.                                                                                                                       |
| 011           | 1111    | 00  | <a href="#">SMULWB</a> , <a href="#">SMULWT</a> - <a href="#">SMULWB</a> variant                                                   |
| 011           | 1111    | 01  | <a href="#">SMULWB</a> , <a href="#">SMULWT</a> - <a href="#">SMULWT</a> variant                                                   |
| 100           | != 1111 | 00  | <a href="#">SMLSD</a> , <a href="#">SMLSDX</a> - <a href="#">SMLSD</a> variant                                                     |
| 100           | != 1111 | 01  | <a href="#">SMLSD</a> , <a href="#">SMLSDX</a> - <a href="#">SMLSDX</a> variant                                                    |
| 100           | -       | 1x  | Unallocated.                                                                                                                       |
| 100           | 1111    | 00  | <a href="#">SMUSD</a> , <a href="#">SMUSDX</a> - <a href="#">SMUSD</a> variant                                                     |
| 100           | 1111    | 01  | <a href="#">SMUSD</a> , <a href="#">SMUSDX</a> - <a href="#">SMUSDX</a> variant                                                    |
| 101           | != 1111 | 00  | <a href="#">SMMLA</a> , <a href="#">SMMLAR</a> - <a href="#">SMMLA</a> variant                                                     |
| 101           | != 1111 | 01  | <a href="#">SMMLA</a> , <a href="#">SMMLAR</a> - <a href="#">SMMLAR</a> variant                                                    |
| 101           | -       | 1x  | Unallocated.                                                                                                                       |
| 101           | 1111    | 00  | <a href="#">SMMUL</a> , <a href="#">SMMULR</a> - <a href="#">SMMUL</a> variant                                                     |
| 101           | 1111    | 01  | <a href="#">SMMUL</a> , <a href="#">SMMULR</a> - <a href="#">SMMULR</a> variant                                                    |
| 110           | -       | 00  | <a href="#">SMMLS</a> , <a href="#">SMMLSR</a> - <a href="#">SMMLS</a> variant                                                     |
| 110           | -       | 01  | <a href="#">SMMLS</a> , <a href="#">SMMLSR</a> - <a href="#">SMMLSR</a> variant                                                    |
| 110           | -       | 1x  | Unallocated.                                                                                                                       |
| 111           | != 1111 | 00  | <a href="#">USADA8</a>                                                                                                             |
| 111           | -       | 01  | Unallocated.                                                                                                                       |
| 111           | -       | 1x  | Unallocated.                                                                                                                       |
| 111           | 1111    | 00  | <a href="#">USAD8</a>                                                                                                              |

### C2.3.9 Long multiply and divide

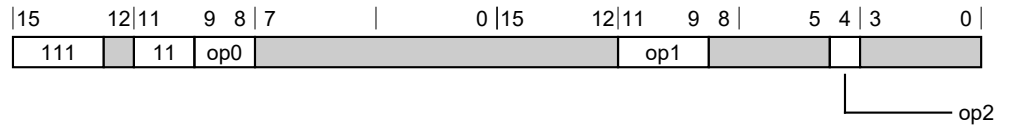
This section describes the encoding of the Long multiply and divide instruction class. The encodings in this section are decoded from [32-bit T32 instruction encoding](#).

|    |    |    |    |    |    |   |   |   |   |     |   |    |    |      |    |      |   |     |   |    |  |
|----|----|----|----|----|----|---|---|---|---|-----|---|----|----|------|----|------|---|-----|---|----|--|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 4   | 3 | 0  | 15 | 12   | 11 | 8    | 7 | 4   | 3 | 0  |  |
| 1  | 1  | 1  | 1  | 1  | 0  | 1 | 1 | 1 | 1 | op1 |   | Rn |    | RdLo |    | RdHi |   | op2 |   | Rm |  |

| Decode fields |         | Instruction page                                                                                                                        |
|---------------|---------|-----------------------------------------------------------------------------------------------------------------------------------------|
| op1           | op2     |                                                                                                                                         |
| 000           | != 0000 | Unallocated.                                                                                                                            |
| 000           | 0000    | <a href="#">SMULL</a>                                                                                                                   |
| 001           | != 1111 | Unallocated.                                                                                                                            |
| 001           | 1111    | <a href="#">SDIV</a>                                                                                                                    |
| 010           | != 0000 | Unallocated.                                                                                                                            |
| 010           | 0000    | <a href="#">UMULL</a>                                                                                                                   |
| 011           | != 1111 | Unallocated.                                                                                                                            |
| 011           | 1111    | <a href="#">UDIV</a>                                                                                                                    |
| 100           | 0000    | <a href="#">SMLAL</a>                                                                                                                   |
| 100           | 0001    | Unallocated.                                                                                                                            |
| 100           | 001x    | Unallocated.                                                                                                                            |
| 100           | 01xx    | Unallocated.                                                                                                                            |
| 100           | 1000    | <a href="#">SMLALBB</a> , <a href="#">SMLALBT</a> , <a href="#">SMLALTB</a> , <a href="#">SMLALTT</a> - <a href="#">SMLALBB</a> variant |
| 100           | 1001    | <a href="#">SMLALBB</a> , <a href="#">SMLALBT</a> , <a href="#">SMLALTB</a> , <a href="#">SMLALTT</a> - <a href="#">SMLALBT</a> variant |
| 100           | 1010    | <a href="#">SMLALBB</a> , <a href="#">SMLALBT</a> , <a href="#">SMLALTB</a> , <a href="#">SMLALTT</a> - <a href="#">SMLALTB</a> variant |
| 100           | 1011    | <a href="#">SMLALBB</a> , <a href="#">SMLALBT</a> , <a href="#">SMLALTB</a> , <a href="#">SMLALTT</a> - <a href="#">SMLALTT</a> variant |
| 100           | 1100    | <a href="#">SMLALD</a> , <a href="#">SMLALDX</a> - <a href="#">SMLALD</a> variant                                                       |
| 100           | 1101    | <a href="#">SMLALD</a> , <a href="#">SMLALDX</a> - <a href="#">SMLALDX</a> variant                                                      |
| 100           | 111x    | Unallocated.                                                                                                                            |
| 101           | 0xxx    | Unallocated.                                                                                                                            |
| 101           | 10xx    | Unallocated.                                                                                                                            |
| 101           | 1100    | <a href="#">SMLS LD</a> , <a href="#">SMLS LD X</a> - <a href="#">SMLS LD</a> variant                                                   |
| 101           | 1101    | <a href="#">SMLS LD</a> , <a href="#">SMLS LD X</a> - <a href="#">SMLS LD X</a> variant                                                 |
| 101           | 111x    | Unallocated.                                                                                                                            |
| 110           | 0000    | <a href="#">UMLAL</a>                                                                                                                   |
| 110           | 0001    | Unallocated.                                                                                                                            |
| 110           | 001x    | Unallocated.                                                                                                                            |
| 110           | 010x    | Unallocated.                                                                                                                            |
| 110           | 0110    | <a href="#">UMAAL</a>                                                                                                                   |
| 110           | 0111    | Unallocated.                                                                                                                            |
| 110           | 1xxx    | Unallocated.                                                                                                                            |
| 111           | -       | Unallocated.                                                                                                                            |

### C2.3.10 Coprocessor and floating-point instructions

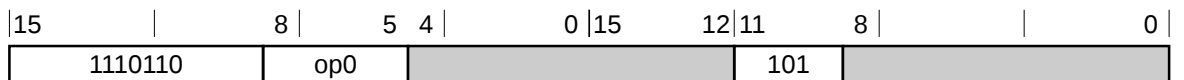
This section describes the encoding of the Coprocessor and floating-point instructions group. The encodings in this section are decoded from [32-bit T32 instruction encoding](#).



| Decode fields |       |     | Decode group or instruction page                    |
|---------------|-------|-----|-----------------------------------------------------|
| op0           | op1   | op2 |                                                     |
| 0x            | 101   | -   | Floating-point load/store and 64-bit register moves |
| 10            | 101   | 0   | Floating-point data-processing                      |
| 10            | 101   | 1   | Floating-point 32-bit register moves                |
| 11            | -     | -   | Unallocated.                                        |
| !=11          | !=101 | -   | Coprocessor                                         |

### Floating-point load/store and 64-bit register moves

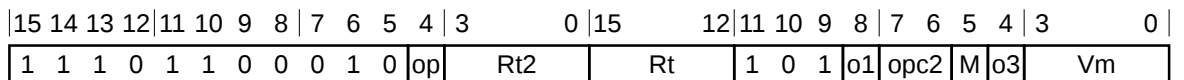
This section describes the encoding of the Floating-point load/store and 64-bit register moves group. The encodings in this section are decoded from [Coprocessor and floating-point instructions](#).



| Decode fields | Decode group or instruction page |
|---------------|----------------------------------|
| op0           |                                  |
| 0000          | Unallocated.                     |
| 0010          | Floating-point 64-bit move       |
| != 00x0       | Floating-point load/store        |

### Floating-point 64-bit move

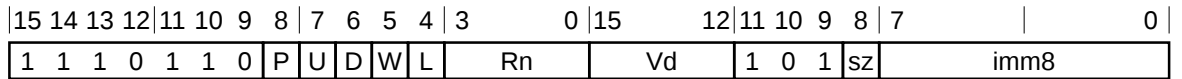
This section describes the encoding of the Floating-point 64-bit move instruction class. The encodings in this section are decoded from [Floating-point load/store and 64-bit register moves](#).



| Decode fields |    |       |    | Instruction page                                                                |
|---------------|----|-------|----|---------------------------------------------------------------------------------|
| op            | o1 | opc2  | o3 |                                                                                 |
| -             | -  | != 00 | -  | Unallocated.                                                                    |
| -             | -  | -     | 0  | Unallocated.                                                                    |
| 0             | 0  | 00    | 1  | VMOV (between two general-purpose registers and two single-precision registers) |
| 0             | 1  | 00    | 1  | VMOV (between two general-purpose registers and a doubleword register)          |
| 1             | 0  | 00    | 1  | VMOV (between two general-purpose registers and two single-precision registers) |
| 1             | 1  | 00    | 1  | VMOV (between two general-purpose registers and a doubleword register)          |

### Floating-point load/store

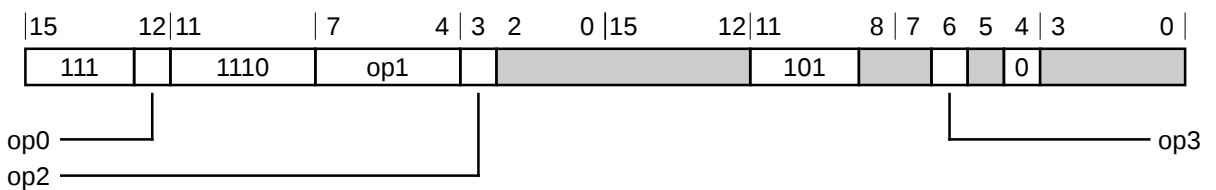
This section describes the encoding of the Floating-point load/store instruction class. The encodings in this section are decoded from [Floating-point load/store and 64-bit register moves](#).



| Decode fields |   |   |   |    |           |                                             | Instruction page |
|---------------|---|---|---|----|-----------|---------------------------------------------|------------------|
| P             | U | W | L | sz | imm8      |                                             |                  |
| 0             | 0 | 1 | 0 | 0  | -         | VLSTM                                       |                  |
| 0             | 0 | 1 | - | 1  | -         | Unallocated.                                |                  |
| 0             | 0 | 1 | 1 | 0  | -         | VLLDM                                       |                  |
| 0             | 1 | - | 0 | 0  | -         | VSTM                                        |                  |
| 0             | 1 | - | 0 | 1  | xxxxxxxx0 | VSTM                                        |                  |
| 0             | 1 | - | 0 | 1  | xxxxxxxx1 | FSTMDBX, FSTMIAX - Increment After variant  |                  |
| 0             | 1 | - | 1 | 0  | -         | VLDM                                        |                  |
| 0             | 1 | - | 1 | 1  | xxxxxxxx0 | VLDM                                        |                  |
| 0             | 1 | - | 1 | 1  | xxxxxxxx1 | FLDMDBX, FLDMIAX - Increment After variant  |                  |
| 1             | - | 0 | 0 | -  | -         | VSTR                                        |                  |
| 1             | - | 0 | 1 | -  | -         | VLDR                                        |                  |
| 1             | 0 | 1 | 0 | 0  | -         | VSTM                                        |                  |
| 1             | 0 | 1 | 0 | 1  | xxxxxxxx0 | VSTM                                        |                  |
| 1             | 0 | 1 | 0 | 1  | xxxxxxxx1 | FSTMDBX, FSTMIAX - Decrement Before variant |                  |
| 1             | 0 | 1 | 1 | 0  | -         | VLDM                                        |                  |
| 1             | 0 | 1 | 1 | 1  | xxxxxxxx0 | VLDM                                        |                  |
| 1             | 0 | 1 | 1 | 1  | xxxxxxxx1 | FLDMDBX, FLDMIAX - Decrement Before variant |                  |
| 1             | 1 | 1 | - | -  | -         | Unallocated.                                |                  |

### Floating-point data-processing

This section describes the encoding of the Floating-point data-processing group. The encodings in this section are decoded from [Coprocessor and floating-point instructions](#).

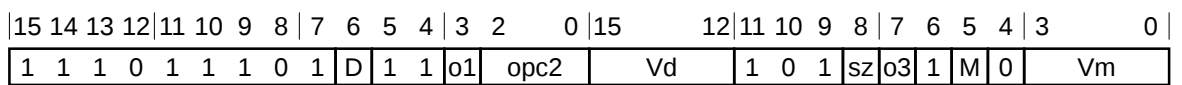


| Decode fields |         |     |     | Decode group or instruction page                 |
|---------------|---------|-----|-----|--------------------------------------------------|
| op0           | op1     | op2 | op3 |                                                  |
| 0             | 1x11    | -   | 1   | Floating-point data-processing (two registers)   |
| 0             | 1x11    | -   | 0   | VMOV (immediate)                                 |
| 0             | != 1x11 | -   | -   | Floating-point data-processing (three registers) |
| 1             | 0xxx    | -   | 0   | VSEL                                             |
| 1             | 0xxx    | -   | 1   | Unallocated.                                     |
| 1             | 1x00    | -   | -   | Floating-point minNum / maxNum                   |
| 1             | 1x01    | -   | -   | Unallocated.                                     |

| Decode fields |      |     |     | Decode group or instruction page           |
|---------------|------|-----|-----|--------------------------------------------|
| op0           | op1  | op2 | op3 |                                            |
| 1             | 1x10 | –   | –   | Unallocated.                               |
| 1             | 1x11 | 0   | –   | Unallocated.                               |
| 1             | 1x11 | 1   | 0   | Unallocated.                               |
| 1             | 1x11 | 1   | 1   | Floating-point directed convert to integer |

### Floating-point data-processing (two registers)

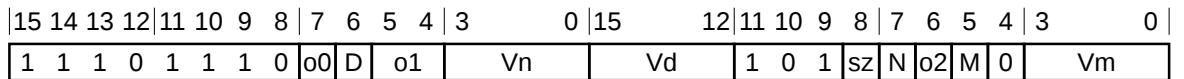
This section describes the encoding of the Floating-point data-processing (two registers) instruction class. The encodings in this section are decoded from [Floating-point data-processing](#).



| Decode fields |      |    | Instruction page                                     |
|---------------|------|----|------------------------------------------------------|
| o1            | opc2 | o3 |                                                      |
| 0             | 000  | 0  | VMOV (register)                                      |
| 0             | 000  | 1  | VABS                                                 |
| 0             | 001  | 0  | VNEG                                                 |
| 0             | 001  | 1  | VSQRT                                                |
| 0             | 010  | 0  | VCVTB                                                |
| 0             | 010  | 1  | VCVTT                                                |
| 0             | 011  | 0  | VCVTB                                                |
| 0             | 011  | 1  | VCVTT                                                |
| 0             | 100  | 0  | VCMP - T1                                            |
| 0             | 100  | 1  | VCMPE - T1                                           |
| 0             | 101  | 0  | VCMP - T2                                            |
| 0             | 101  | 1  | VCMPE - T2                                           |
| 0             | 110  | 0  | VRINTR                                               |
| 0             | 110  | 1  | VRINTZ                                               |
| 0             | 111  | 0  | VRINTX                                               |
| 0             | 111  | 1  | VCVT (between double-precision and single-precision) |
| 1             | 000  | –  | VCVT (integer to floating-point)                     |
| 1             | 001  | –  | Unallocated.                                         |
| 1             | 01x  | –  | VCVT (between floating-point and fixed-point)        |
| 1             | 100  | 0  | VCVTR                                                |
| 1             | 100  | 1  | VCVT (floating-point to integer)                     |
| 1             | 101  | 0  | VCVTR                                                |
| 1             | 101  | 1  | VCVT (floating-point to integer)                     |
| 1             | 11x  | –  | VCVT (between floating-point and fixed-point)        |

### Floating-point data-processing (three registers)

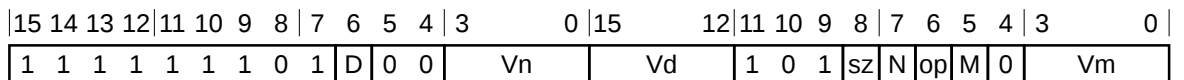
This section describes the encoding of the Floating-point data-processing (three registers) instruction class. The encodings in this section are decoded from [Floating-point data-processing](#).



| Decode fields |    |    | Instruction page |
|---------------|----|----|------------------|
| o0            | o1 | o2 |                  |
| 0             | 00 | 0  | VMLA             |
| 0             | 00 | 1  | VMLS             |
| 0             | 01 | 0  | VNMLS            |
| 0             | 01 | 1  | VNMLA            |
| 0             | 10 | 0  | VMUL             |
| 0             | 10 | 1  | VNMUL            |
| 0             | 11 | 0  | VADD             |
| 0             | 11 | 1  | VSUB             |
| 1             | 00 | 0  | VDIV             |
| 1             | 01 | 0  | VFNMS            |
| 1             | 01 | 1  | VFNMA            |
| 1             | 10 | 0  | VFMA             |
| 1             | 10 | 1  | VFMS             |

**Floating-point minNum / maxNum**

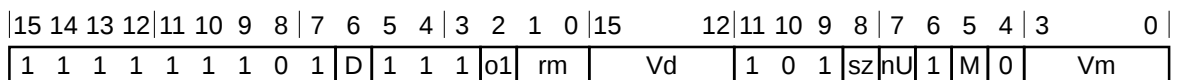
This section describes the encoding of the Floating-point minNum / maxNum instruction class. The encodings in this section are decoded from [Floating-point data-processing](#).



| Decode fields |  | Instruction page |
|---------------|--|------------------|
| op            |  |                  |
| 0             |  | VMAXNM           |
| 1             |  | VMINNM           |

**Floating-point directed convert to integer**

This section describes the encoding of the Floating-point directed convert to integer instruction class. The encodings in this section are decoded from [Floating-point data-processing](#).

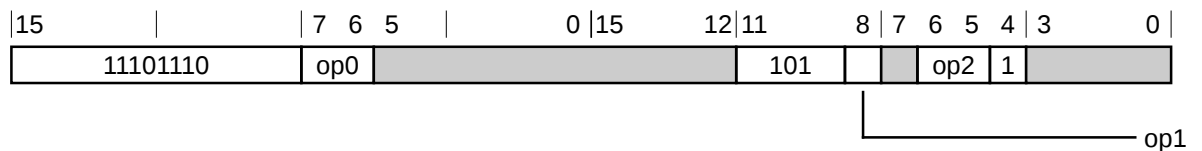


| Decode fields |    | Instruction page |
|---------------|----|------------------|
| o1            | rm |                  |
| 0             | 00 | VRINTA           |
| 0             | 01 | VRINTN           |

| Decode fields |    | Instruction page |
|---------------|----|------------------|
| o1            | rm |                  |
| 0             | 10 | VRINTP           |
| 0             | 11 | VRINTM           |
| 1             | 00 | VCVTA            |
| 1             | 01 | VCVTN            |
| 1             | 10 | VCVTP            |
| 1             | 11 | VCVTM            |

### Floating-point 32-bit register moves

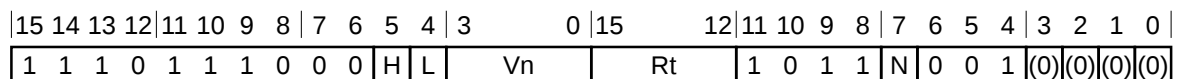
This section describes the encoding of the Floating-point 32-bit register moves group. The encodings in this section are decoded from [Coprocesor and floating-point instructions](#).



| Decode fields |     |       | Decode group or instruction page                      |
|---------------|-----|-------|-------------------------------------------------------|
| op0           | op1 | op2   |                                                       |
| 00            | 1   | 00    | <a href="#">Floating-point 32-bit move doubleword</a> |
| 00            | 1   | != 00 | Unallocated.                                          |
| != 00         | 1   | -     | Unallocated.                                          |
| -             | 0   | -     | <a href="#">Floating-point 32-bit move</a>            |

### Floating-point 32-bit move doubleword

This section describes the encoding of the Floating-point 32-bit move doubleword instruction class. The encodings in this section are decoded from [Floating-point 32-bit register moves](#).

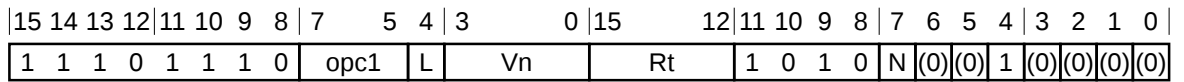


| Decode fields | Instruction page                                                                      |
|---------------|---------------------------------------------------------------------------------------|
| <b>L</b>      |                                                                                       |
| 0             | <a href="#">VMOV (single general-purpose register to half of doubleword register)</a> |
| 1             | <a href="#">VMOV (half of doubleword register to single general-purpose register)</a> |

### Floating-point 32-bit move

This section describes the encoding of the Floating-point 32-bit move instruction class. The encodings in this section are decoded from [Floating-point 32-bit register moves](#).

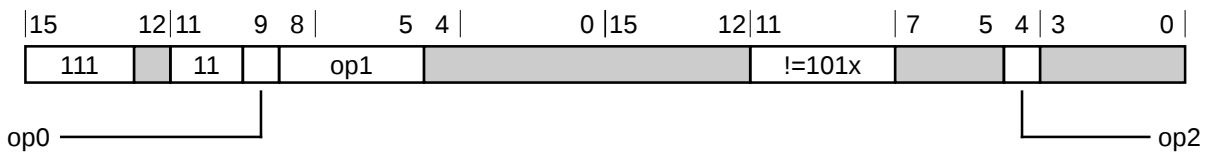




| Decode fields | Instruction page |                                                                       |
|---------------|------------------|-----------------------------------------------------------------------|
| <b>opc1</b>   | <b>L</b>         |                                                                       |
| 000           | –                | VMOV (between general-purpose register and single-precision register) |
| 001           | –                | Unallocated.                                                          |
| 01x           | –                | Unallocated.                                                          |
| 10x           | –                | Unallocated.                                                          |
| 110           | –                | Unallocated.                                                          |
| 111           | 0                | VMSR                                                                  |
| 111           | 1                | VMRS                                                                  |

### Coprocessor

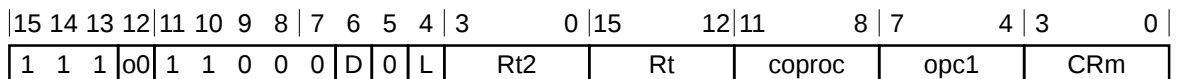
This section describes the encoding of the Coprocessor group. The encodings in this section are decoded from [Coprocessor and floating-point instructions](#).



| Decode fields |         |     | Decode group or instruction page |
|---------------|---------|-----|----------------------------------|
| op0           | op1     | op2 |                                  |
| 0             | 00x0    | –   | Coprocessor 64-bit move          |
| 0             | != 00x0 | –   | Coprocessor load/store registers |
| 1             | 0xxx    | 0   | CDP, CDP2                        |
| 1             | 0xxx    | 1   | Coprocessor 32-bit move          |

### Coprocessor 64-bit move

This section describes the encoding of the Coprocessor 64-bit move instruction class. The encodings in this section are decoded from [Coprocessor](#).



| Decode fields |   |   | Instruction page |
|---------------|---|---|------------------|
| o0            | D | L |                  |
| 0             | 0 | – | Unallocated.     |
| 0             | 1 | 0 | MCRR, MCRR2 - T1 |
| 0             | 1 | 1 | MRRC, MRRC2 - T1 |
| 1             | 0 | – | Unallocated.     |
| 1             | 1 | 0 | MCRR, MCRR2 - T2 |

| Decode fields |   |   | Instruction page                 |
|---------------|---|---|----------------------------------|
| o0            | D | L |                                  |
| 1             | 1 | 1 | <a href="#">MRRC, MRRC2</a> - T2 |

### Coprocessor load/store registers

This section describes the encoding of the Coprocessor load/store registers instruction class. The encodings in this section are decoded from [Coprocessor](#).

|    |    |    |    |    |    |   |   |   |   |   |   |    |     |        |      |    |   |   |   |
|----|----|----|----|----|----|---|---|---|---|---|---|----|-----|--------|------|----|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3  | 0   | 15     | 12   | 11 | 8 | 7 | 0 |
| 1  | 1  | 1  | o0 | 1  | 1  | 0 | P | U | D | W | L | Rn | CRd | coproc | imm8 |    |   |   |   |

| Decode fields |        |   | Instruction page |                                          |
|---------------|--------|---|------------------|------------------------------------------|
| o0            | P:U:W  | L | Rn               |                                          |
| 0             | != 000 | 1 | 1111             | <a href="#">LDC, LDC2 (literal)</a> - T1 |
| 0             | 0x1    | 0 | -                | <a href="#">STC, STC2</a>                |
| 0             | 0x1    | 1 | != 1111          | <a href="#">LDC, LDC2 (immediate)</a>    |
| 0             | 010    | 0 | -                | <a href="#">STC, STC2</a>                |
| 0             | 010    | 1 | != 1111          | <a href="#">LDC, LDC2 (immediate)</a>    |
| 0             | 1x0    | 0 | -                | <a href="#">STC, STC2</a>                |
| 0             | 1x0    | 1 | != 1111          | <a href="#">LDC, LDC2 (immediate)</a>    |
| 0             | 1x1    | 0 | -                | <a href="#">STC, STC2</a>                |
| 0             | 1x1    | 1 | != 1111          | <a href="#">LDC, LDC2 (immediate)</a>    |
| 1             | != 000 | 1 | 1111             | <a href="#">LDC, LDC2 (literal)</a> - T2 |
| 1             | 0x1    | 0 | -                | <a href="#">STC, STC2</a>                |
| 1             | 0x1    | 1 | != 1111          | <a href="#">LDC, LDC2 (immediate)</a>    |
| 1             | 010    | 0 | -                | <a href="#">STC, STC2</a>                |
| 1             | 010    | 1 | != 1111          | <a href="#">LDC, LDC2 (immediate)</a>    |
| 1             | 1x0    | 0 | -                | <a href="#">STC, STC2</a>                |
| 1             | 1x0    | 1 | != 1111          | <a href="#">LDC, LDC2 (immediate)</a>    |
| 1             | 1x1    | 0 | -                | <a href="#">STC, STC2</a>                |
| 1             | 1x1    | 1 | != 1111          | <a href="#">LDC, LDC2 (immediate)</a>    |

### Coprocessor 32-bit move

This section describes the encoding of the Coprocessor 32-bit move instruction class. The encodings in this section are decoded from [Coprocessor](#).

|    |    |    |    |    |    |   |   |      |   |     |    |        |      |    |     |   |   |   |   |   |   |
|----|----|----|----|----|----|---|---|------|---|-----|----|--------|------|----|-----|---|---|---|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7    | 5 | 4   | 3  | 0      | 15   | 12 | 11  | 8 | 7 | 5 | 4 | 3 | 0 |
| 1  | 1  | 1  | o0 | 1  | 1  | 1 | 0 | opc1 | L | CRn | Rt | coproc | opc2 | 1  | CRm |   |   |   |   |   |   |

| Decode fields |   | Instruction page               |
|---------------|---|--------------------------------|
| o0            | L |                                |
| 0             | 0 | <a href="#">MCR, MCR2</a> - T1 |
| 0             | 1 | <a href="#">MRC, MRC2</a> - T1 |
| 1             | 0 | <a href="#">MCR, MCR2</a> - T2 |
| 1             | 1 | <a href="#">MRC, MRC2</a> - T2 |

## C2.4 Alphabetical list of instructions

Every Armv8-M instruction is listed in this section. See [Chapter C1 Instruction Set Overview on page 305](#) for the format of the instruction descriptions.

### C2.4.1 ADC (immediate)

Add with Carry (immediate). Add with Carry (immediate) adds an immediate value and the carry flag value to a register value, and writes the result to the destination register. It can optionally update the condition flags based on the result.

#### T1

Armv8-M Main Extension only

|    |    |    |    |    |    |   |   |   |   |   |   |    |   |   |   |    |      |    |    |    |    |   |   |   |      |   |   |   |   |   |   |  |
|----|----|----|----|----|----|---|---|---|---|---|---|----|---|---|---|----|------|----|----|----|----|---|---|---|------|---|---|---|---|---|---|--|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3  | 2 | 1 | 0 | 15 | 14   | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6    | 5 | 4 | 3 | 2 | 1 | 0 |  |
| 1  | 1  | 1  | 1  | 0  | i  | 0 | 1 | 0 | 1 | 0 | S | Rn |   |   |   | 0  | imm3 |    |    |    | Rd |   |   |   | imm8 |   |   |   |   |   |   |  |

#### ADC variant

Applies when **S == 0**.

ADC{<c>}{<q>} {<Rd>}, {<Rn>, #<const>

#### ADCS variant

Applies when **S == 1**.

ADCS{<c>}{<q>} {<Rd>}, {<Rn>, #<const>

#### Decode for this encoding

```

1 if !HaveMainExt() then UNDEFINED;
2 d = UInt(Rd); n = UInt(Rn); setflags = (S == '1'); imm32 = T32ExpandImm(i:imm3:imm8);
3 if d IN {13,15} || n IN {13,15} then UNPREDICTABLE;

```

#### Assembler symbols for all encodings

|         |                                                                                                                                                                                                |
|---------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <c>     | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                                                                                                                       |
| <q>     | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                                                                                                                       |
| <Rd>    | Is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as <Rn>.                                                                         |
| <Rn>    | Is the general-purpose source register, encoded in the "Rn" field.                                                                                                                             |
| <const> | Is an immediate value derived from the 12-bit immediate that is encoded in the 'i:imm3:imm8' field. See <a href="#">C1.5 Modified immediate constants</a> on page 324 for the range of values. |

#### Operation for all encodings

```

1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 (result, carry, overflow) = AddWithCarry(R[n], imm32, APSR.C);
4 R[d] = result;
5 if setflags then
6 APSR.N = result[31];
7 APSR.Z = IsZeroBit(result);
8 APSR.C = carry;
9 APSR.V = overflow;

```

## C2.4.2 ADC (register)

Add with Carry (register). Add with Carry (register) adds a register value, the carry flag value, and an optionally-shifted register value, and writes the result to the destination register. It can optionally update the condition flags based on the result.

### T1

Armv8-M

|    |    |    |    |    |    |   |   |   |   |    |   |   |   |   |     |
|----|----|----|----|----|----|---|---|---|---|----|---|---|---|---|-----|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5  | 4 | 3 | 2 | 1 | 0   |
| 0  | 1  | 0  | 0  | 0  | 0  | 0 | 1 | 0 | 1 | Rm |   |   |   |   | Rdn |

#### T1 variant

```
ADC<c>{<q>} {<Rdn>, } <Rdn>, <Rm>
 // Inside IT block
ADCS{<q>} {<Rdn>, } <Rdn>, <Rm>
 // Outside IT block
```

#### Decode for this encoding

```
1 d = UInt(Rdn); n = UInt(Rdn); m = UInt(Rm); setflags = !InITBlock();
2 (shift_t, shift_n) = (SRTYPE_LSL, 0);
```

### T2

Armv8-M Main Extension only

|    |    |    |    |    |    |   |   |   |   |   |   |    |     |      |    |      |    |    |    |    |    |   |   |    |   |   |   |   |   |   |   |
|----|----|----|----|----|----|---|---|---|---|---|---|----|-----|------|----|------|----|----|----|----|----|---|---|----|---|---|---|---|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3  | 2   | 1    | 0  | 15   | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7  | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 1  | 1  | 1  | 0  | 1  | 0  | 1 | 1 | 0 | 1 | 0 | S | Rn | (0) | imm3 | Rd | imm2 |    |    |    |    |    |   |   | Rm |   |   |   |   |   |   |   |

|  
sr\_type

#### ADC, rotate right with extend variant

Applies when **S == 0** && **imm3 == 000** && **imm2 == 00** && **sr\_type == 11**.

```
ADC{<c>}{<q>} {<Rd>, } <Rn>, <Rm>, RRX
```

#### ADC, shift or rotate by value variant

Applies when **S == 0** && **!(imm3 == 000 && imm2 == 00 && sr\_type == 11)**.

```
ADC<c>.W {<Rd>, } <Rn>, <Rm>
 // Inside IT block, and <Rd>, <Rn>, <Rm> can be represented in T1
ADC{<c>}{<q>} {<Rd>, } <Rn>, <Rm> {, <shift> #<amount>}
```

#### ADCS, rotate right with extend variant

Applies when **S == 1** && **imm3 == 000** && **imm2 == 00** && **sr\_type == 11**.

```
ADCS{<c>}{<q>} {<Rd>, } <Rn>, <Rm>, RRX
```

#### ADCS, shift or rotate by value variant

Applies when **S == 1** && **!(imm3 == 000 && imm2 == 00 && sr\_type == 11)**.

```
ADCS.W {<Rd>, } <Rn>, <Rm>
// Outside IT block, and <Rd>, <Rn>, <Rm> can be represented in T1
ADCS{<c>}{<q>} {<Rd>, } <Rn>, <Rm> {, <shift> #<amount>}
```

### Decode for this encoding

```
1 if !HaveMainExt() then UNDEFINED;
2 d = UInt(Rd); n = UInt(Rn); m = UInt(Rm); setflags = (S == '1');
3 (shift_t, shift_n) = DecodeImmShift(sr_type, imm3:imm2);
4 if d IN {13,15} || n IN {13,15} || m IN {13,15} then UNPREDICTABLE;
```

### Assembler symbols for all encodings

<c> See [C1.2.5 Standard assembler syntax fields](#) on page 309.

<q> See [C1.2.5 Standard assembler syntax fields](#) on page 309.

<Rdn> Is the first general-purpose source register and the destination register, encoded in the "Rdn" field.

<Rd> Is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as <Rn>.

<Rn> Is the first general-purpose source register, encoded in the "Rn" field.

<Rm> Is the second general-purpose source register, encoded in the "Rm" field.

<shift> Is the type of shift to be applied to the second source register, encoded in the "sr\_type" field. It can have the following values:

- LSL when sr\_type = 00
- LSR when sr\_type = 01
- ASR when sr\_type = 10
- ROR when sr\_type = 11

<amount> Is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR) encoded in the "imm3:imm2" field as <amount> modulo 32.

### Operation for all encodings

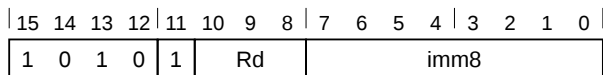
```
1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 shifted = Shift(R[m], shift_t, shift_n, APSR.C);
4 (result, carry, overflow) = AddWithCarry(R[n], shifted, APSR.C);
5 R[d] = result;
6 if setflags then
7 APSR.N = result[31];
8 APSR.Z = IsZeroBit(result);
9 APSR.C = carry;
10 APSR.V = overflow;
```

### C2.4.3 ADD (SP plus immediate)

Add to SP (immediate). ADD (SP plus immediate) adds an immediate value to the SP value, and writes the result to the destination register.

#### T1

Armv8-M



#### T1 variant

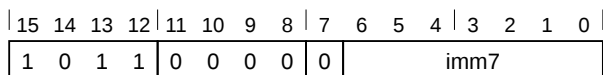
ADD{<c>}{<q>} <Rd>, SP, #<imm8>

#### Decode for this encoding

```
1 d = UInt(Rd); setflags = FALSE; imm32 = ZeroExtend(imm8:'00', 32);
```

#### T2

Armv8-M



#### T2 variant

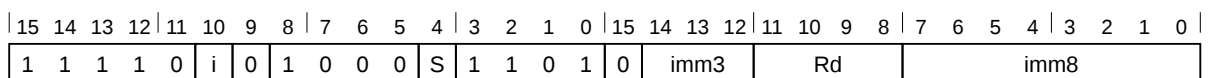
ADD{<c>}{<q>} {SP,} SP, #<imm7>

#### Decode for this encoding

```
1 d = 13; setflags = FALSE; imm32 = ZeroExtend(imm7:'00', 32);
```

#### T3

Armv8-M Main Extension only



#### ADD variant

Applies when **S == 0**.

```
ADD{<c>}.W {<Rd>,} SP, #<const>
// <Rd>, <const> can be represented in T1 or T2
ADD{<c>}{<q>} {<Rd>,} SP, #<const>
```

#### ADDS variant

Applies when **S == 1 && Rd != 1111**.

```
ADDS{<c>}{<q>} {<Rd>,} SP, #<const>
```

#### Decode for this encoding





### C2.4.4 ADD (SP plus register)

Add to SP (register). ADD (SP plus register) adds an optionally-shifted register value to the SP value, and writes the result to the destination register.

#### T1

Armv8-M

|    |    |    |    |    |    |   |   |    |   |   |   |   |     |   |   |
|----|----|----|----|----|----|---|---|----|---|---|---|---|-----|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7  | 6 | 5 | 4 | 3 | 2   | 1 | 0 |
| 0  | 1  | 0  | 0  | 0  | 1  | 0 | 0 | DM | 1 | 1 | 0 | 1 | Rdm |   |   |

#### T1 variant

ADD{<c>}{<q>} {<Rdm>}, SP, <Rdm>

#### Decode for this encoding

```
1 d = UInt(DM:Rdm); m = UInt(DM:Rdm); setflags = FALSE;
2 if d == 15 && InITBlock() && !LastInITBlock() then UNPREDICTABLE;
3 (shift_t, shift_n) = (SRType_LSL, 0);
```

#### T2

Armv8-M

|    |    |    |    |    |    |   |   |   |            |   |   |   |   |   |   |
|----|----|----|----|----|----|---|---|---|------------|---|---|---|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6          | 5 | 4 | 3 | 2 | 1 | 0 |
| 0  | 1  | 0  | 0  | 0  | 1  | 0 | 0 | 1 | Rm != 1101 |   |   | 1 | 0 | 1 |   |

#### T2 variant

ADD{<c>}{<q>} {SP}, SP, <Rm>

#### Decode for this encoding

```
1 if Rm == '1101' then SEE "encoding T1";
2 d = 13; m = UInt(Rm); setflags = FALSE;
3 (shift_t, shift_n) = (SRType_LSL, 0);
```

#### T3

Armv8-M Main Extension only

|    |    |    |    |    |    |   |   |   |   |   |   |   |   |   |   |     |      |    |    |    |    |   |      |   |    |   |   |   |   |   |   |
|----|----|----|----|----|----|---|---|---|---|---|---|---|---|---|---|-----|------|----|----|----|----|---|------|---|----|---|---|---|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | 15  | 14   | 13 | 12 | 11 | 10 | 9 | 8    | 7 | 6  | 5 | 4 | 3 | 2 | 1 | 0 |
| 1  | 1  | 1  | 0  | 1  | 0  | 1 | 1 | 0 | 0 | 0 | S | 1 | 1 | 0 | 1 | (0) | imm3 |    |    | Rd |    |   | imm2 |   | Rm |   |   |   |   |   |   |

|  
sr\_type

#### ADD, rotate right with extend variant

Applies when **S == 0 && imm3 == 000 && imm2 == 00 && sr\_type == 11**.

ADD{<c>}{<q>} {<Rd>}, SP, <Rm>, RRX

#### ADD, shift or rotate by value variant

Applies when **S == 0 && !(imm3 == 000 && imm2 == 00 && sr\_type == 11)**.

ADD{<c>}.W {<Rd>}, SP, <Rm>

// <Rd>, <Rm> can be represented in T1 or T2

ADD{<c>}{<q>} {<Rd>}, SP, <Rm> {, <shift> #<amount>}

### ADDS, rotate right with extend variant

Applies when **S == 1 && imm3 == 000 && Rd != 1111 && imm2 == 00 && sr\_type == 11**.

ADDS{<c>}{<q>} {<Rd>}, SP, <Rm>, RRX

### ADDS, shift or rotate by value variant

Applies when **S == 1 && !(imm3 == 000 && imm2 == 00 && sr\_type == 11) && Rd != 1111**.

ADDS{<c>}{<q>} {<Rd>}, SP, <Rm> {, <shift> #<amount>}

### Decode for this encoding

```

1 if Rd == '1111' && S == '1' then SEE "CMN (register)";
2 if !HaveMainExt() then UNDEFINED;
3 d = UInt(Rd); m = UInt(Rm); setflags = (S == '1');
4 (shift_t, shift_n) = DecodeImmShift(sr_type, imm3:imm2);
5 if d == 13 && (shift_t != SRTYPE_LSL || shift_n > 3) then UNPREDICTABLE;
6 if (d == 15 && S == '0') || m IN {13,15} then UNPREDICTABLE;
```

### Assembler symbols for all encodings

|          |                                                                                                                                                                                                                                                                                                |
|----------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <c>      | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                                                                                                                                                                                                                       |
| <q>      | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                                                                                                                                                                                                                       |
| <Rdm>    | Is the general-purpose destination and second source register, encoded in the "Rdm" field. If omitted, this register is the SP. Arm deprecates using the PC as the destination register, but if the PC is used, the instruction is a simple branch to the address calculated by the operation. |
| <Rd>     | Is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the SP.                                                                                                                                                                                   |
| <Rm>     | For encoding T2: is the second general-purpose source register, encoded in the "Rm" field. The PC can be used, but this is deprecated.<br>For encoding T3: is the second general-purpose source register, encoded in the "Rm" field.                                                           |
| <shift>  | Is the type of shift to be applied to the second source register, encoded in the "sr_type" field. It can have the following values:<br>LSL when sr_type = 00<br>LSR when sr_type = 01<br>ASR when sr_type = 10<br>ROR when sr_type = 11                                                        |
| <amount> | Is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR) encoded in the "imm3:imm2" field as <amount> modulo 32.                                                                                                                           |

### Operation for all encodings

```

1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 shifted = Shift(R[m], shift_t, shift_n, APSR.C);
4 (result, carry, overflow) = AddWithCarry(SP, shifted, '0');
5 if d == 15 then
6 ALUWritePC(result); // setflags is always FALSE here
7 else
8 RSPCheck[d] = result;
9 if setflags then
10 APSR.N = result[31];
11 APSR.Z = IsZeroBit(result);
12 APSR.C = carry;
13 APSR.V = overflow;
```

### C2.4.5 ADD (immediate)

Add (immediate). Add (immediate) adds an immediate value to a register value, and writes the result to the destination register. It can optionally update the condition flags based on the result.

#### T1

Armv8-M

|    |    |    |    |    |    |   |      |   |   |    |   |   |    |   |   |
|----|----|----|----|----|----|---|------|---|---|----|---|---|----|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8    | 7 | 6 | 5  | 4 | 3 | 2  | 1 | 0 |
| 0  | 0  | 0  | 1  | 1  | 1  | 0 | imm3 |   |   | Rn |   |   | Rd |   |   |

#### T1 variant

```
ADD<c>{<q>} <Rd>, <Rn>, #<imm3>
// Inside IT block
ADDS{<q>} <Rd>, <Rn>, #<imm3>
// Outside IT block
```

#### Decode for this encoding

```
1 d = UInt(Rd); n = UInt(Rn); setflags = !InITBlock(); imm32 = ZeroExtend(imm3, 32);
```

#### T2

Armv8-M

|    |    |    |    |    |     |   |   |      |   |   |   |   |   |   |   |
|----|----|----|----|----|-----|---|---|------|---|---|---|---|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10  | 9 | 8 | 7    | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 0  | 0  | 1  | 1  | 0  | Rdn |   |   | imm8 |   |   |   |   |   |   |   |

#### T2 variant

```
ADD<c>{<q>} <Rdn>, #<imm8>
// Inside IT block, and <Rdn>, <imm8> can be represented in T1
ADD<c>{<q>} {<Rdn>, } <Rdn>, #<imm8>
// Inside IT block, and <Rdn>, <imm8> cannot be represented in T1
ADDS{<q>} <Rdn>, #<imm8>
// Outside IT block, and <Rdn>, <imm8> can be represented in T1
ADDS{<q>} {<Rdn>, } <Rdn>, #<imm8>
// Outside IT block, and <Rdn>, <imm8> cannot be represented in T1
```

#### Decode for this encoding

```
1 d = UInt(Rdn); n = UInt(Rdn); setflags = !InITBlock(); imm32 = ZeroExtend(imm8, 32);
```

#### T3

Armv8-M Main Extension only

|    |    |    |    |    |    |   |   |   |   |   |   |            |   |   |   |      |    |    |    |    |    |      |   |   |   |   |   |   |   |   |   |
|----|----|----|----|----|----|---|---|---|---|---|---|------------|---|---|---|------|----|----|----|----|----|------|---|---|---|---|---|---|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3          | 2 | 1 | 0 | 15   | 14 | 13 | 12 | 11 | 10 | 9    | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 1  | 1  | 1  | 1  | 0  | i  | 0 | 1 | 0 | 0 | 0 | S | Rn != 1101 |   |   | 0 | imm3 |    |    | Rd |    |    | imm8 |   |   |   |   |   |   |   |   |   |

#### ADD variant

Applies when S == 0.

```
ADD<c>.W {<Rd>, } <Rn>, #<const>
// Inside IT block, and <Rd>, <Rn>, <const> can be represented in T1 or T2
ADD{<c>}{<q>} {<Rd>, } <Rn>, #<const>
```

### ADDS variant

Applies when **S == 1 && Rd != 1111**.

```
ADDS.W {<Rd>}, {<Rn>}, #<const>
// Outside IT block, and <Rd>, <Rn>, <const> can be represented in T1 or T2
ADDS{<c>}{<q>} {<Rd>}, {<Rn>}, #<const>
```

### Decode for this encoding

```
1 if Rd == '1111' && S == '1' then SEE "CMN (immediate)";
2 if Rn == '1101' then SEE "ADD (SP plus immediate)"
3 if !HaveMainExt() then UNDEFINED;
4 d = UInt(Rd); n = UInt(Rn); setflags = (S == '1'); imm32 = T32ExpandImm(i:imm3:imm8);
5 if d == 13 || (d == 15 && S == '0') || n == 15 then UNPREDICTABLE;
```

### T4

Armv8-M Main Extension only

|    |    |    |    |    |    |   |   |   |   |   |   |            |   |      |    |      |    |    |    |    |    |   |   |   |   |   |   |   |   |   |   |
|----|----|----|----|----|----|---|---|---|---|---|---|------------|---|------|----|------|----|----|----|----|----|---|---|---|---|---|---|---|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3          | 2 | 1    | 0  | 15   | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 1  | 1  | 1  | 1  | 0  | i  | 1 | 0 | 0 | 0 | 0 | 0 | Rn != 11x1 | 0 | imm3 | Rd | imm8 |    |    |    |    |    |   |   |   |   |   |   |   |   |   |   |

### T4 variant

```
ADD{<c>}{<q>} {<Rd>}, {<Rn>}, #<imm12>
// <imm12> cannot be represented in T1, T2, or T3
ADDW{<c>}{<q>} {<Rd>}, {<Rn>}, #<imm12>
// <imm12> can be represented in T1, T2, or T3
```

### Decode for this encoding

```
1 if Rn == '1111' then SEE ADR;
2 if Rn == '1101' then SEE "ADD (SP plus immediate)"
3 if !HaveMainExt() then UNDEFINED;
4 d = UInt(Rd); n = UInt(Rn); setflags = FALSE; imm32 = ZeroExtend(i:imm3:imm8, 32);
5 if d IN {13,15} then UNPREDICTABLE;
```

### Assembler symbols for all encodings

|         |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                        |
|---------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <c>     | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                                                                                                                                                                                                                                                                                                                                                                                                                               |
| <q>     | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                                                                                                                                                                                                                                                                                                                                                                                                                               |
| <Rdn>   | Is the general-purpose source and destination register, encoded in the "Rdn" field.                                                                                                                                                                                                                                                                                                                                                                                                                    |
| <imm8>  | Is an 8-bit unsigned immediate, in the range 0 to 255, encoded in the "imm8" field.                                                                                                                                                                                                                                                                                                                                                                                                                    |
| <Rd>    | Is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as <Rn>.                                                                                                                                                                                                                                                                                                                                                                                 |
| <Rn>    | For encoding T1: is the general-purpose source register, encoded in the "Rn" field.<br>For encoding T3: is the general-purpose source register, encoded in the "Rn" field. If the SP is used, see <a href="#">C2.4.3 ADD (SP plus immediate)</a> on page 371.<br>For encoding T4: is the general-purpose source register, encoded in the "Rn" field. If the SP is used, see <a href="#">C2.4.3 ADD (SP plus immediate)</a> on page 371. If the PC is used, see <a href="#">C2.4.8 ADR</a> on page 383. |
| <imm3>  | Is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the "imm3" field.                                                                                                                                                                                                                                                                                                                                                                                                                       |
| <imm12> | Is a 12-bit unsigned immediate, in the range 0 to 4095, encoded in the "i:imm3:imm8" field.                                                                                                                                                                                                                                                                                                                                                                                                            |
| <const> | Is an immediate value derived from the 12-bit immediate that is encoded in the 'i:imm3:imm8' field. See <a href="#">C1.5 Modified immediate constants</a> on page 324 for the range of values.                                                                                                                                                                                                                                                                                                         |

### Operation for all encodings

```
1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 (result, carry, overflow) = AddWithCarry(R[n], imm32, '0');
4 R[d] = result;
5 if setflags then
6 APSR.N = result[31];
7 APSR.Z = IsZeroBit(result);
8 APSR.C = carry;
9 APSR.V = overflow;
```

### C2.4.6 ADD (immediate, to PC)

Add to PC. Add to PC adds an immediate value to the Align(PC, 4) value to form a PC-relative address, and writes the result to the destination register. Arm recommends that, where possible, software avoids using this alias.

This instruction is a pseudo-instruction of the [ADR](#) instruction. This means that:

- The encodings in this description are named to match the encodings of [ADR](#).
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of [ADR](#) gives the operational pseudocode for this instruction.

#### T1

Armv8-M

|    |    |    |    |    |    |   |   |   |      |   |   |   |   |   |   |  |
|----|----|----|----|----|----|---|---|---|------|---|---|---|---|---|---|--|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6    | 5 | 4 | 3 | 2 | 1 | 0 |  |
| 1  | 0  | 1  | 0  | 0  | Rd |   |   |   | imm8 |   |   |   |   |   |   |  |

#### T1 variant

ADD{<c>}{<q>} <Rd>, PC, #<imm8>

is equivalent to

ADR{<c>}{<q>} <Rd>, <label>

and is never the preferred disassembly.

#### T3

Armv8-M Main Extension only

|    |    |    |    |    |    |   |   |   |   |   |   |   |   |   |   |    |      |    |    |    |    |      |   |   |   |   |   |   |   |   |   |
|----|----|----|----|----|----|---|---|---|---|---|---|---|---|---|---|----|------|----|----|----|----|------|---|---|---|---|---|---|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | 15 | 14   | 13 | 12 | 11 | 10 | 9    | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 1  | 1  | 1  | 1  | 0  | i  | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 0  | imm3 | Rd |    |    |    | imm8 |   |   |   |   |   |   |   |   |   |

#### T3 variant

ADDW{<c>}{<q>} <Rd>, PC, #<imm12>  
 // <Rd>, <imm12> can be represented in T1

is equivalent to

ADR{<c>}{<q>} <Rd>, <label>

and is never the preferred disassembly.

#### T3 variant

ADD{<c>}{<q>} <Rd>, PC, #<imm12>

is equivalent to

ADR{<c>}{<q>} <Rd>, <label>

and is never the preferred disassembly.

#### Assembler symbols for all encodings

- <c> See [C1.2.5 Standard assembler syntax fields](#) on page 309.  
 <q> See [C1.2.5 Standard assembler syntax fields](#) on page 309.  
 <Rd> Is the general-purpose destination register, encoded in the "Rd" field.

- <label> For encoding T1: the label of an instruction or literal data item whose address is to be loaded into <Rd>. The assembler calculates the required value of the offset from the [Align](#) (PC, 4) value of the ADR instruction to this label. Permitted values of the size of the offset are multiples of 4 in the range 0 to 1020.  
For encoding T2 and T3: the label of an instruction or literal data item whose address is to be loaded into <Rd>. The assembler calculates the required value of the offset from the [Align](#) (PC, 4) value of the ADR instruction to this label. If the offset is zero or positive, encoding T3 is used, with `imm32` equal to the offset. If the offset is negative, encoding T2 is used, with `imm32` equal to the size of the offset. That is, the use of encoding T2 indicates that the required offset is minus the value of `imm32`. Permitted values of the size of the offset are 0-4095.
- <imm8> Is an unsigned immediate, a multiple of 4, in the range 0 to 1020, encoded in the "imm8" field as <imm8>/4.
- <imm12> Is a 12-bit unsigned immediate, in the range 0 to 4095, encoded in the "i:imm3:imm8" field.

### Operation for all encodings

The description of [ADR](#) gives the operational pseudocode for this instruction.

### C2.4.7 ADD (register)

Add (register). ADD (register) adds a register value and an optionally-shifted register value, and writes the result to the destination register. It can optionally update the condition flags based on the result.

#### T1

Armv8-M

|    |    |    |    |    |    |   |   |    |   |   |    |   |    |   |   |
|----|----|----|----|----|----|---|---|----|---|---|----|---|----|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7  | 6 | 5 | 4  | 3 | 2  | 1 | 0 |
| 0  | 0  | 0  | 1  | 1  | 0  | 0 |   | Rm |   |   | Rn |   | Rd |   |   |

#### T1 variant

```
ADD<c>{<q>} <Rd>, <Rn>, <Rm>
 // Inside IT block
ADDS{<q>} {<Rd>, } <Rn>, <Rm>
 // Outside IT block
```

#### Decode for this encoding

```
1 d = UInt(Rd); n = UInt(Rn); m = UInt(Rm); setflags = !InITBlock();
2 (shift_t, shift_n) = (SRTYPE_LSL, 0);
```

#### T2

Armv8-M

|    |    |    |    |    |    |   |   |    |            |   |   |     |   |   |   |
|----|----|----|----|----|----|---|---|----|------------|---|---|-----|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7  | 6          | 5 | 4 | 3   | 2 | 1 | 0 |
| 0  | 1  | 0  | 0  | 0  | 1  | 0 | 0 | DN | Rm != 1101 |   |   | Rdn |   |   |   |

#### T2 variant

Applies when **!(DN == 1 && Rdn == 101)**.

```
ADD<c>{<q>} <Rdn>, <Rm>
 // Preferred syntax, Inside IT block
ADD{<c>}{<q>} {<Rdn>, } <Rdn>, <Rm>
```

#### Decode for this encoding

```
1 if (DN:Rdn) == '1101' || Rm == '1101' then SEE "ADD (SP plus register)"
2 d = UInt(DN:Rdn); n = UInt(DN:Rdn); m = UInt(Rm); setflags = FALSE;
3 (shift_t, shift_n) = (SRTYPE_LSL, 0);
4 if d == 15 && InITBlock() && !LastInITBlock() then UNPREDICTABLE;
5 if d == 15 && m == 15 then UNPREDICTABLE;
```

#### T3

Armv8-M Main Extension only

|    |    |    |    |    |    |   |   |   |   |   |   |            |   |     |      |    |    |      |    |    |    |   |   |   |   |   |   |   |   |   |   |
|----|----|----|----|----|----|---|---|---|---|---|---|------------|---|-----|------|----|----|------|----|----|----|---|---|---|---|---|---|---|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3          | 2 | 1   | 0    | 15 | 14 | 13   | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 1  | 1  | 1  | 0  | 1  | 0  | 1 | 1 | 0 | 0 | 0 | S | Rn != 1101 |   | (0) | imm3 | Rd |    | imm2 |    |    | Rm |   |   |   |   |   |   |   |   |   |   |

|  
sr\_type

#### ADD, rotate right with extend variant

Applies when **S == 0 && imm3 == 000 && imm2 == 00 && sr\_type == 11**.



ADD{<c>}{<q>} {<Rd>}, <Rn>, <Rm>, RRX

#### ADD, shift or rotate by value variant

Applies when **S == 0 && !(imm3 == 000 && imm2 == 00 && sr\_type == 11)**.

```
ADD<c>.W {<Rd>}, <Rn>, <Rm>
// Inside IT block, and <Rd>, <Rn>, <Rm> can be represented in T1 or T2
ADD{<c>}{<q>} {<Rd>}, <Rn>, <Rm> {, <shift> #<amount>}
```

#### ADDS, rotate right with extend variant

Applies when **S == 1 && imm3 == 000 && Rd != 1111 && imm2 == 00 && sr\_type == 11**.

ADDS{<c>}{<q>} {<Rd>}, <Rn>, <Rm>, RRX

#### ADDS, shift or rotate by value variant

Applies when **S == 1 && !(imm3 == 000 && imm2 == 00 && sr\_type == 11) && Rd != 1111**.

```
ADDS.W {<Rd>}, <Rn>, <Rm>
// Outside IT block, and <Rd>, <Rn>, <Rm> can be represented in T1 or T2
ADDS{<c>}{<q>} {<Rd>}, <Rn>, <Rm> {, <shift> #<amount>}
```

#### Decode for this encoding

```
1 if Rd == '1111' && S == '1' then SEE "CMN (register)";
2 if Rn == '1101' then SEE "ADD (SP plus register)"
3 if !HaveMainExt() then UNDEFINED;
4 d = UInt(Rd); n = UInt(Rn); m = UInt(Rm); setflags = (S == '1');
5 (shift_t, shift_n) = DecodeImmShift(sr_type, imm3:imm2);
6 if d == 13 || (d == 15 && S == '0') || n == 15 || m IN {13,15} then UNPREDICTABLE;
```

#### Assembler symbols for all encodings

|       |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                              |
|-------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <c>   | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                     |
| <q>   | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                     |
| <Rdn> | Is the general-purpose source and destination register, encoded in the "DN:Rdn" field. If the PC is used, the instruction is a branch to the address calculated by the operation. This is a simple branch. The assembler language allows <Rdn> to be specified once or twice in the assembler syntax. When used inside an IT block, and <Rdn> and <Rm> are in the range R0 to R7, <Rdn> must be specified once so that encoding T2 is preferred to encoding T1. In all other cases there is no difference in behavior when <Rdn> is specified once or twice. |
| <Rd>  | For encoding T1: is the general-purpose destination register, encoded in the "Rd" field. When used inside an IT block, <Rd> must be specified. When used outside an IT block, <Rd> is optional, and: <ul style="list-style-type: none"> <li>- If omitted, this register is the same as &lt;Rn&gt;.</li> <li>- If present, encoding T1 is preferred to encoding T2.</li> </ul> For encoding T3: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as <Rn>.                                        |
| <Rn>  | For encoding T1: is the first general-purpose source register, encoded in the "Rn" field.<br>For encoding T3: is the first general-purpose source register, encoded in the "Rn" field. If the SP is used, see <a href="#">C2.4.4 ADD (SP plus register)</a> on page 373.                                                                                                                                                                                                                                                                                     |
| <Rm>  | For encoding T1 and T3: is the second general-purpose source register, encoded in the "Rm" field.<br>For encoding T2: is the second general-purpose source register, encoded in the "Rm" field. The PC can be used.                                                                                                                                                                                                                                                                                                                                          |

<shift> Is the type of shift to be applied to the second source register, encoded in the "sr\_type" field. It can have the following values:

- LSL when sr\_type = 00
- LSR when sr\_type = 01
- ASR when sr\_type = 10
- ROR when sr\_type = 11

<amount> Is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR) encoded in the "imm3:imm2" field as <amount> modulo 32.

### Operation for all encodings

```

1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 shifted = Shift(R[m], shift_t, shift_n, APSR.C);
4 (result, carry, overflow) = AddWithCarry(R[n], shifted, '0');
5 if d == 15 then
6 ALUWritePC(result); // setflags is always FALSE here
7 else
8 R[d] = result;
9 if setflags then
10 APSR.N = result[31];
11 APSR.Z = IsZeroBit(result);
12 APSR.C = carry;
13 APSR.V = overflow;

```

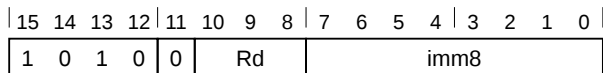
### C2.4.8 ADR

Form PC-relative address. Address to Register adds an immediate value to the PC value, and writes the result to the destination register.

This instruction is used by the pseudo-instructions [ADD \(immediate, to PC\)](#) and [SUB \(immediate, from PC\)](#).

#### T1

*Armv8-M*



#### T1 variant

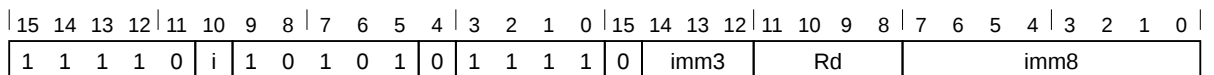
ADR{<c>}{<q>} <Rd>, <label>

#### Decode for this encoding

```
1 d = UInt(Rd); imm32 = ZeroExtend(imm8:'00', 32); add = TRUE;
```

#### T2

*Armv8-M Main Extension only*



#### T2 variant

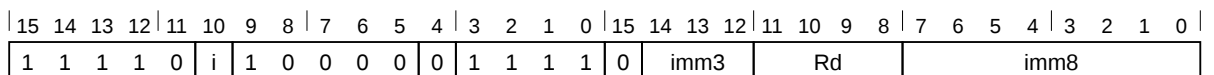
ADR{<c>}{<q>} <Rd>, <label>

#### Decode for this encoding

```
1 if !HaveMainExt() then UNDEFINED;
2 d = UInt(Rd); imm32 = ZeroExtend(i:imm3:imm8, 32); add = FALSE;
3 if d IN {13,15} then UNPREDICTABLE;
```

#### T3

*Armv8-M Main Extension only*



#### T3 variant

ADR{<c>}.W <Rd>, <label>  
 // <Rd>, <label> can be presented in T1  
 ADR{<c>}{<q>} <Rd>, <label>

#### Decode for this encoding

```
1 if !HaveMainExt() then UNDEFINED;
2 d = UInt(Rd); imm32 = ZeroExtend(i:imm3:imm8, 32); add = TRUE;
3 if d IN {13,15} then UNPREDICTABLE;
```

#### Alias conditions

| Alias or pseudo-instruction              | preferred when                             |
|------------------------------------------|--------------------------------------------|
| <a href="#">ADD (immediate, to PC)</a>   | Never                                      |
| <a href="#">SUB (immediate, from PC)</a> | <code>i:imm3:imm8 == '000000000000'</code> |

### Assembler symbols for all encodings

- `<c>` See [C1.2.5 Standard assembler syntax fields](#) on page 309.
- `<q>` See [C1.2.5 Standard assembler syntax fields](#) on page 309.
- `<Rd>` Is the general-purpose destination register, encoded in the "Rd" field.
- `<label>` For encoding T1: the label of an instruction or literal data item whose address is to be loaded into `<Rd>`. The assembler calculates the required value of the offset from the [Align \(PC, 4\)](#) value of the ADR instruction to this label. Permitted values of the size of the offset are multiples of 4 in the range 0 to 1020.  
For encoding T2 and T3: the label of an instruction or literal data item whose address is to be loaded into `<Rd>`. The assembler calculates the required value of the offset from the [Align \(PC, 4\)](#) value of the ADR instruction to this label. If the offset is zero or positive, encoding T3 is used, with `imm32` equal to the offset. If the offset is negative, encoding T2 is used, with `imm32` equal to the size of the offset. That is, the use of encoding T2 indicates that the required offset is minus the value of `imm32`. Permitted values of the size of the offset are 0-4095.

### Operation for all encodings

```
1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 result = if add then (Align(PC, 4) + imm32) else (Align(PC, 4) - imm32);
4 R[d] = result;
```

## C2.4.9 AND (immediate)

Bitwise AND (immediate). AND (immediate) performs a bitwise AND of a register value and an immediate value, and writes the result to the destination register.

### T1

Armv8-M Main Extension only

|    |    |    |    |    |    |   |   |   |   |   |   |    |   |   |   |    |      |    |    |    |    |   |   |   |      |   |   |   |   |   |   |
|----|----|----|----|----|----|---|---|---|---|---|---|----|---|---|---|----|------|----|----|----|----|---|---|---|------|---|---|---|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3  | 2 | 1 | 0 | 15 | 14   | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6    | 5 | 4 | 3 | 2 | 1 | 0 |
| 1  | 1  | 1  | 1  | 0  | i  | 0 | 0 | 0 | 0 | 0 | S | Rn |   |   |   | 0  | imm3 |    |    |    | Rd |   |   |   | imm8 |   |   |   |   |   |   |

### AND variant

Applies when **S == 0**.

AND{<c>}{<q>} {<Rd>}, {<Rn>}, #<const>

### ANDS variant

Applies when **S == 1 && Rd != 1111**.

ANDS{<c>}{<q>} {<Rd>}, {<Rn>}, #<const>

### Decode for this encoding

```

1 if Rd == '1111' && S == '1' then SEE "TST (immediate)";
2 if !HaveMainExt() then UNDEFINED;
3 d = UInt(Rd); n = UInt(Rn); setflags = (S == '1');
4 (imm32, carry) = T32ExpandImm_C(i:imm3:imm8, APSR.C);
5 if d == 13 || (d == 15 && S == '0') || n IN {13,15} then UNPREDICTABLE;

```

### Assembler symbols for all encodings

<c> See [C1.2.5 Standard assembler syntax fields](#) on page 309.  
 <q> See [C1.2.5 Standard assembler syntax fields](#) on page 309.  
 <Rd> Is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as <Rn>.  
 <Rn> Is the general-purpose source register, encoded in the "Rn" field.  
 <const> Is an immediate value derived from the 12-bit immediate that is encoded in the 'i:imm3:imm8' field. See [C1.5 Modified immediate constants](#) on page 324 for the range of values.

### Operation for all encodings

```

1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 result = R[n] AND imm32;
4 R[d] = result;
5 if setflags then
6 APSR.N = result[31];
7 APSR.Z = IsZeroBit(result);
8 APSR.C = carry;
9 // APSR.V unchanged

```

## C2.4.10 AND (register)

Bitwise AND (register). AND (register) performs a bitwise AND of a register value and an optionally-shifted register value, and writes the result to the destination register. It can optionally update the condition flags based on the result.

### T1

Armv8-M

|    |    |    |    |    |    |   |   |   |   |    |   |   |   |   |     |
|----|----|----|----|----|----|---|---|---|---|----|---|---|---|---|-----|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5  | 4 | 3 | 2 | 1 | 0   |
| 0  | 1  | 0  | 0  | 0  | 0  | 0 | 0 | 0 | 0 | Rm |   |   |   |   | Rdn |

#### T1 variant

```
AND<c>{<q>} {<Rdn>}, <Rdn>, <Rm>
 // Inside IT block
ANDS{<q>} {<Rdn>}, <Rdn>, <Rm>
 // Outside IT block
```

#### Decode for this encoding

```
1 d = UInt(Rdn); n = UInt(Rdn); m = UInt(Rm); setflags = !InITBlock();
2 (shift_t, shift_n) = (SRTYPE_LSL, 0);
```

### T2

Armv8-M Main Extension only

|    |    |    |    |    |    |   |   |   |   |   |   |    |     |      |    |      |    |    |    |    |    |   |    |   |   |   |   |   |   |   |   |
|----|----|----|----|----|----|---|---|---|---|---|---|----|-----|------|----|------|----|----|----|----|----|---|----|---|---|---|---|---|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3  | 2   | 1    | 0  | 15   | 14 | 13 | 12 | 11 | 10 | 9 | 8  | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 1  | 1  | 1  | 0  | 1  | 0  | 1 | 0 | 0 | 0 | 0 | S | Rn | (0) | imm3 | Rd | imm2 |    |    |    |    |    |   | Rm |   |   |   |   |   |   |   |   |

|  
sr\_type

#### AND, rotate right with extend variant

Applies when **S == 0** && **imm3 == 000** && **imm2 == 00** && **sr\_type == 11**.

```
AND{<c>}{<q>} {<Rd>}, <Rn>, <Rm>, RRX
```

#### AND, shift or rotate by value variant

Applies when **S == 0** && **!(imm3 == 000 && imm2 == 00 && sr\_type == 11)**.

```
AND<c>.W {<Rd>}, <Rn>, <Rm>
 // Inside IT block, and <Rd>, <Rn>, <Rm> can be represented in T1
AND{<c>}{<q>} {<Rd>}, <Rn>, <Rm> {, <shift> #<amount>}
```

#### ANDS, rotate right with extend variant

Applies when **S == 1** && **imm3 == 000** && **Rd != 1111** && **imm2 == 00** && **sr\_type == 11**.

```
ANDS{<c>}{<q>} {<Rd>}, <Rn>, <Rm>, RRX
```

#### ANDS, shift or rotate by value variant

Applies when **S == 1** && **!(imm3 == 000 && imm2 == 00 && sr\_type == 11)** && **Rd != 1111**.

```
ANDS.W {<Rd>,} <Rn>, <Rm>
// Outside IT block, and <Rd>, <Rn>, <Rm> can be represented in T1
ANDS{<c>}{<q>} {<Rd>,} <Rn>, <Rm> {, <shift> #<amount>}
```

### Decode for this encoding

```
1 if Rd == '1111' && S == '1' then SEE "TST (register)";
2 if !HaveMainExt() then UNDEFINED;
3 d = UInt(Rd); n = UInt(Rn); m = UInt(Rm); setflags = (S == '1');
4 (shift_t, shift_n) = DecodeImmShift(sr_type, imm3:imm2);
5 if d == 13 || (d == 15 && S == '0') || n IN {13,15} || m IN {13,15} then UNPREDICTABLE;
```

### Assembler symbols for all encodings

<c> See [C1.2.5 Standard assembler syntax fields](#) on page 309.

<q> See [C1.2.5 Standard assembler syntax fields](#) on page 309.

<Rdn> Is the first general-purpose source register and the destination register, encoded in the "Rdn" field.

<Rd> Is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as <Rn>.

<Rn> Is the first general-purpose source register, encoded in the "Rn" field.

<Rm> Is the second general-purpose source register, encoded in the "Rm" field.

<shift> Is the type of shift to be applied to the second source register, encoded in the "sr\_type" field. It can have the following values:

|     |                   |
|-----|-------------------|
| LSL | when sr_type = 00 |
| LSR | when sr_type = 01 |
| ASR | when sr_type = 10 |
| ROR | when sr_type = 11 |

<amount> Is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR) encoded in the "imm3:imm2" field as <amount> modulo 32.

### Operation for all encodings

```
1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 (shifted, carry) = Shift_C(R[m], shift_t, shift_n, APSR.C);
4 result = R[n] AND shifted;
5 R[d] = result;
6 if setflags then
7 APSR.N = result[31];
8 APSR.Z = IsZeroBit(result);
9 APSR.C = carry;
10 // APSR.V unchanged
```

### C2.4.11 ASR (immediate)

Arithmetic Shift Right (immediate). Arithmetic Shift Right (immediate) shifts a register value right by an immediate number of bits, shifting in copies of its sign bit, and writes the result to the destination register.

This instruction is an alias of the [MOV \(register\)](#) instruction. This means that:

- The encodings in this description are named to match the encodings of [MOV \(register\)](#).
- The description of [MOV \(register\)](#) gives the operational pseudocode for this instruction.

#### T2

*Armv8-M Main Extension only*

|       |    |    |    |         |    |   |   |      |   |   |   |    |   |    |   |
|-------|----|----|----|---------|----|---|---|------|---|---|---|----|---|----|---|
| 15    | 14 | 13 | 12 | 11      | 10 | 9 | 8 | 7    | 6 | 5 | 4 | 3  | 2 | 1  | 0 |
| 0 0 0 |    |    |    | op = 10 |    |   |   | imm5 |   |   |   | Rm |   | Rd |   |

#### T2 variant

```
ASR<c>{<q>} {<Rd>}, <Rm>, #<imm>
// Inside IT block
```

**is equivalent to**

```
MOV<c>{<q>} <Rd>, <Rm>, ASR #<imm>
```

**and is the preferred disassembly when `InITBlock()`.**

#### T3

*Armv8-M Main Extension only*

|           |    |    |    |     |    |   |   |         |   |   |   |         |   |   |   |     |    |    |    |      |    |   |   |    |   |   |   |              |   |   |   |    |  |  |  |
|-----------|----|----|----|-----|----|---|---|---------|---|---|---|---------|---|---|---|-----|----|----|----|------|----|---|---|----|---|---|---|--------------|---|---|---|----|--|--|--|
| 15        | 14 | 13 | 12 | 11  | 10 | 9 | 8 | 7       | 6 | 5 | 4 | 3       | 2 | 1 | 0 | 15  | 14 | 13 | 12 | 11   | 10 | 9 | 8 | 7  | 6 | 5 | 4 | 3            | 2 | 1 | 0 |    |  |  |  |
| 1 1 1 0 1 |    |    |    | 0 1 |    |   |   | 0 0 1 0 |   |   |   | 1 1 1 1 |   |   |   | (0) |    |    |    | imm3 |    |   |   | Rd |   |   |   | imm2         |   |   |   | Rm |  |  |  |
|           |    |    |    |     |    |   |   |         |   |   |   | S = 0   |   |   |   |     |    |    |    |      |    |   |   |    |   |   |   | sr_type = 10 |   |   |   |    |  |  |  |

#### MOV, shift or rotate by value variant

```
ASR<c>.W {<Rd>}, <Rm>, #<imm>
// Inside IT block, and <Rd>, <Rm>, <imm> can be represented in T2
```

**is equivalent to**

```
MOV{<c>}{<q>} <Rd>, <Rm>, ASR #<imm>
```

**and is always the preferred disassembly.**

#### MOV, shift or rotate by value variant

```
ASR{<c>}{<q>} {<Rd>}, <Rm>, #<imm>
```

**is equivalent to**

```
MOV{<c>}{<q>} <Rd>, <Rm>, ASR #<imm>
```

**and is always the preferred disassembly.**

#### Assembler symbols for all encodings

<c> See [C1.2.5 Standard assembler syntax fields](#) on page 309.

<q> See [C1.2.5 Standard assembler syntax fields](#) on page 309.



|       |                                                                                                                                                                                                                                 |
|-------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <Rd>  | Is the general-purpose destination register, encoded in the "Rd" field.                                                                                                                                                         |
| <Rm>  | Is the general-purpose source register, encoded in the "Rm" field.                                                                                                                                                              |
| <imm> | For encoding T2: is the shift amount, in the range 1 to 32, encoded in the "imm5" field as <imm> modulo 32.<br>For encoding T3: is the shift amount, in the range 1 to 32, encoded in the "imm3:imm2" field as <imm> modulo 32. |

### Operation for all encodings

The description of [MOV \(register\)](#) gives the operational pseudocode for this instruction.

## C2.4.12 ASR (register)

Arithmetic Shift Right (register). Arithmetic Shift Right (register) shifts a register value right by a variable number of bits, shifting in copies of its sign bit, and writes the result to the destination registers. The variable number of bits is read from the bottom byte of a register.

This instruction is an alias of the [MOV, MOVS \(register-shifted register\)](#) instruction. This means that:

- The encodings in this description are named to match the encodings of [MOV, MOVS \(register-shifted register\)](#).
- The description of [MOV, MOVS \(register-shifted register\)](#) gives the operational pseudocode for this instruction.

### T1

*Armv8-M Main Extension only*

|    |    |    |    |    |    |           |   |    |   |   |   |     |   |   |   |
|----|----|----|----|----|----|-----------|---|----|---|---|---|-----|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9         | 8 | 7  | 6 | 5 | 4 | 3   | 2 | 1 | 0 |
| 0  | 1  | 0  | 0  | 0  | 0  | op = 0100 |   | Rs |   |   |   | Rdm |   |   |   |

#### Arithmetic shift right variant

```
ASR<c>{<q>} {<Rdm>, } <Rdm>, <Rs>
// Inside IT block
```

#### is equivalent to

```
MOV<c>{<q>} <Rdm>, <Rdm>, ASR <Rs>
```

**and is the preferred disassembly when `InITBlock()`.**

### T2

*Armv8-M Main Extension only*

|    |    |    |    |    |    |   |   |   |                       |   |   |   |    |   |   |    |    |    |    |    |    |   |   |   |   |   |   |   |    |   |   |  |
|----|----|----|----|----|----|---|---|---|-----------------------|---|---|---|----|---|---|----|----|----|----|----|----|---|---|---|---|---|---|---|----|---|---|--|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6                     | 5 | 4 | 3 | 2  | 1 | 0 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2  | 1 | 0 |  |
| 1  | 1  | 1  | 1  | 1  | 0  | 1 | 0 | 0 | sr_type = 10<br>S = 0 |   |   |   | Rm |   |   |    | 1  | 1  | 1  | 1  | Rd |   |   |   | 0 | 0 | 0 | 0 | Rs |   |   |  |

#### Non flag setting variant

```
ASR<c>.W {<Rd>, } <Rm>, <Rs>
// Inside IT block, and <Rd>, <Rm>, <sr_type>, <Rs> can be represented in T1
```

#### is equivalent to

```
MOV{<c>}{<q>} <Rd>, <Rm>, ASR <Rs>
```

**and is always the preferred disassembly.**

#### Non flag setting variant

```
ASR{<c>}{<q>} {<Rd>, } <Rm>, <Rs>
```

#### is equivalent to

```
MOV{<c>}{<q>} <Rd>, <Rm>, ASR <Rs>
```

**and is always the preferred disassembly.**

### Assembler symbols for all encodings

|       |                                                                                                                       |
|-------|-----------------------------------------------------------------------------------------------------------------------|
| <c>   | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                                              |
| <q>   | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                                              |
| <Rdm> | Is the first general-purpose source register and the destination register, encoded in the "Rdm" field.                |
| <Rd>  | Is the general-purpose destination register, encoded in the "Rd" field.                                               |
| <Rm>  | Is the first general-purpose source register, encoded in the "Rm" field.                                              |
| <Rs>  | Is the second general-purpose source register holding a shift amount in its bottom 8 bits, encoded in the "Rs" field. |

### Operation for all encodings

The description of [MOV, MOVS \(register-shifted register\)](#) gives the operational pseudocode for this instruction.

### C2.4.13 ASRS (immediate)

Arithmetic Shift Right, Setting flags (immediate). Arithmetic Shift Right, Setting flags (immediate) shifts a register value right by an immediate number of bits, shifting in copies of its sign bit, writes the result to the destination register, and updates the condition flags based on the result.

This instruction is an alias of the [MOV \(register\)](#) instruction. This means that:

- The encodings in this description are named to match the encodings of [MOV \(register\)](#).
- The description of [MOV \(register\)](#) gives the operational pseudocode for this instruction.

#### T2

*Armv8-M*

|    |    |    |         |      |    |   |   |    |   |   |    |   |   |   |   |
|----|----|----|---------|------|----|---|---|----|---|---|----|---|---|---|---|
| 15 | 14 | 13 | 12      | 11   | 10 | 9 | 8 | 7  | 6 | 5 | 4  | 3 | 2 | 1 | 0 |
| 0  | 0  | 0  | op = 10 | imm5 |    |   |   | Rm |   |   | Rd |   |   |   |   |

#### T2 variant

```
ASRS{<q>} {<Rd>}, <Rm>, #<imm>
// Outside IT block
```

**is equivalent to**

```
MOVS{<q>} <Rd>, <Rm>, ASR #<imm>
```

**and is the preferred disassembly when !InITBlock().**

#### T3

*Armv8-M Main Extension only*

|       |    |    |    |    |    |   |   |   |   |   |   |              |   |   |   |     |      |    |    |    |      |    |   |   |   |   |   |   |   |   |   |
|-------|----|----|----|----|----|---|---|---|---|---|---|--------------|---|---|---|-----|------|----|----|----|------|----|---|---|---|---|---|---|---|---|---|
| 15    | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3            | 2 | 1 | 0 | 15  | 14   | 13 | 12 | 11 | 10   | 9  | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 1     | 1  | 1  | 0  | 1  | 0  | 1 | 0 | 0 | 0 | 1 | 0 | 1            | 1 | 1 | 1 | (0) | imm3 | Rd |    |    | imm2 | Rm |   |   |   |   |   |   |   |   |   |
| S = 1 |    |    |    |    |    |   |   |   |   |   |   | sr_type = 10 |   |   |   |     |      |    |    |    |      |    |   |   |   |   |   |   |   |   |   |

#### MOVS, shift or rotate by value variant

```
ASRS.W {<Rd>}, <Rm>, #<imm>
// Outside IT block, and <Rd>, <Rm>, <imm> can be represented in T2
```

**is equivalent to**

```
MOVS{<c>}{<q>} <Rd>, <Rm>, ASR #<imm>
```

**and is always the preferred disassembly.**

#### MOVS, shift or rotate by value variant

```
ASRS{<c>}{<q>} {<Rd>}, <Rm>, #<imm>
```

**is equivalent to**

```
MOVS{<c>}{<q>} <Rd>, <Rm>, ASR #<imm>
```

**and is always the preferred disassembly.**

#### Assembler symbols for all encodings

<c> See [C1.2.5 Standard assembler syntax fields](#) on page 309.

|       |                                                                                                                                                                                                                                 |
|-------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <q>   | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                                                                                                                                                        |
| <Rd>  | Is the general-purpose destination register, encoded in the "Rd" field.                                                                                                                                                         |
| <Rm>  | Is the general-purpose source register, encoded in the "Rm" field.                                                                                                                                                              |
| <imm> | For encoding T2: is the shift amount, in the range 1 to 32, encoded in the "imm5" field as <imm> modulo 32.<br>For encoding T3: is the shift amount, in the range 1 to 32, encoded in the "imm3:imm2" field as <imm> modulo 32. |

### Operation for all encodings

The description of [MOV \(register\)](#) gives the operational pseudocode for this instruction.

### C2.4.14 ASRS (register)

Arithmetic Shift Right, Setting flags (register). Arithmetic Shift Right, Setting flags (register) shifts a register value right by a variable number of bits, shifting in copies of its sign bit, writes the result to the destination register, and updates the condition flags based on the result. The variable number of bits is read from the bottom byte of a register.

This instruction is an alias of the [MOV, MOVS \(register-shifted register\)](#) instruction. This means that:

- The encodings in this description are named to match the encodings of [MOV, MOVS \(register-shifted register\)](#).
- The description of [MOV, MOVS \(register-shifted register\)](#) gives the operational pseudocode for this instruction.

#### T1

Armv8-M

|    |    |    |    |    |    |           |   |   |   |    |   |   |     |   |   |
|----|----|----|----|----|----|-----------|---|---|---|----|---|---|-----|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9         | 8 | 7 | 6 | 5  | 4 | 3 | 2   | 1 | 0 |
| 0  | 1  | 0  | 0  | 0  | 0  | op = 0100 |   |   |   | Rs |   |   | Rdm |   |   |

#### Arithmetic shift right variant

```
ASRS{<q>} {<Rdm>, } <Rdm>, <Rs>
 // Outside IT block
```

is equivalent to

```
MOVS{<q>} <Rdm>, <Rdm>, ASR <Rs>
```

and is the preferred disassembly when `!InITBlock()`.

#### T2

Armv8-M Main Extension only

|    |    |    |    |    |    |   |   |   |   |   |   |    |   |   |   |    |    |    |    |    |    |   |   |   |   |    |   |   |   |   |   |
|----|----|----|----|----|----|---|---|---|---|---|---|----|---|---|---|----|----|----|----|----|----|---|---|---|---|----|---|---|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3  | 2 | 1 | 0 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5  | 4 | 3 | 2 | 1 | 0 |
| 1  | 1  | 1  | 1  | 1  | 0  | 1 | 0 | 0 |   |   |   | Rm |   |   | 1 | 1  | 1  | 1  | Rd |    |    | 0 | 0 | 0 | 0 | Rs |   |   |   |   |   |

$sr\_type = 10$   
 $S = 1$

#### Flag setting variant

```
ASRS.W {<Rd>, } <Rm>, <Rs>
 // Outside IT block, and <Rd>, <Rm>, <sr_type>, <Rs> can be represented in T1
```

is equivalent to

```
MOVS{<c>}{<q>} <Rd>, <Rm>, ASR <Rs>
```

and is always the preferred disassembly.

#### Flag setting variant

```
ASRS{<c>}{<q>} {<Rd>, } <Rm>, <Rs>
```

is equivalent to

```
MOVS{<c>}{<q>} <Rd>, <Rm>, ASR <Rs>
```

and is always the preferred disassembly.

### Assembler symbols for all encodings

|       |                                                                                                                       |
|-------|-----------------------------------------------------------------------------------------------------------------------|
| <c>   | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                                              |
| <q>   | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                                              |
| <Rdm> | Is the first general-purpose source register and the destination register, encoded in the "Rdm" field.                |
| <Rd>  | Is the general-purpose destination register, encoded in the "Rd" field.                                               |
| <Rm>  | Is the first general-purpose source register, encoded in the "Rm" field.                                              |
| <Rs>  | Is the second general-purpose source register holding a shift amount in its bottom 8 bits, encoded in the "Rs" field. |

### Operation for all encodings

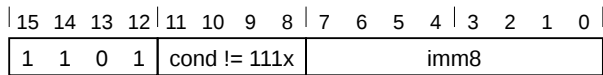
The description of [MOV, MOVS \(register-shifted register\)](#) gives the operational pseudocode for this instruction.

### C2.4.15 B

Branch. Branch causes a branch to a target address.

#### T1

Armv8-M



#### T1 variant

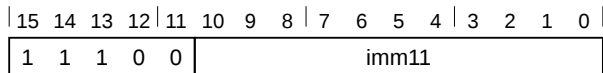
B<c>{<q>} <label>  
 // Not permitted in IT block

#### Decode for this encoding

```
1 if cond == '1110' then SEE UDF;
2 if cond == '1111' then SEE SVC;
3 imm32 = SignExtend(imm8:'0', 32);
4 if InITBlock() then UNPREDICTABLE;
```

#### T2

Armv8-M



#### T2 variant

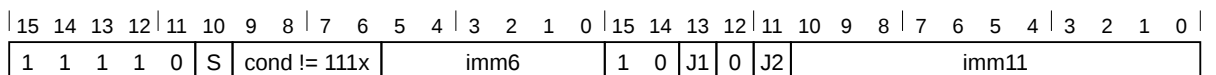
B{<c>}{<q>} <label>  
 // Outside or last in IT block

#### Decode for this encoding

```
1 imm32 = SignExtend(imm11:'0', 32);
2 if InITBlock() && !LastInITBlock() then UNPREDICTABLE;
```

#### T3

Armv8-M Main Extension only



#### T3 variant

B<c>.W <label>  
 // Not permitted in IT block, and <label> can be represented in T1  
 B<c>{<q>} <label>  
 // Not permitted in IT block

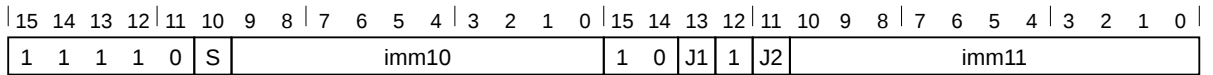
#### Decode for this encoding

```
1 if cond[3:1] == '111' then SEE "Related encodings";
2 if !HaveMainExt() then UNDEFINED;
3 imm32 = SignExtend(S:J2:J1:imm6:imm11:'0', 32);
4 if InITBlock() then UNPREDICTABLE;
```



## T4

Armv8-M



### T4 variant

```
B{<c>}.W <label>
 // <label> can be represented in T2
B{<c>}{<q>} <label>
```

### Decode for this encoding

```
1 I1 = NOT(J1 EOR S); I2 = NOT(J2 EOR S); imm32 = SignExtend(S:I1:I2:imm10:imm11:'0', 32);
2 if InITBlock() && !LastInITBlock() then UNPREDICTABLE;
```

### Assembler symbols for all encodings

<c> For encoding T1: see [C1.2.5 Standard assembler syntax fields](#) on page 309. Must not be AL or omitted.  
 For encoding T2 and T4: see [C1.2.5 Standard assembler syntax fields](#) on page 309.  
 For encoding T3: see [C1.2.5 Standard assembler syntax fields](#) on page 309. <c> must not be AL or omitted.

<q> See [C1.2.5 Standard assembler syntax fields](#) on page 309.

<label> For encoding T1: the label of the instruction that is to be branched to. The assembler calculates the required value of the offset from the PC value of the B instruction to this label, then selects an encoding that sets `imm32` to that offset. Permitted offsets are even numbers in the range -256 to 254.  
 For encoding T2: the label of the instruction that is to be branched to. The assembler calculates the required value of the offset from the PC value of the B instruction to this label, then selects an encoding that sets `imm32` to that offset. Permitted offsets are even numbers in the range -2048 to 2046.  
 For encoding T3: the label of the instruction that is to be branched to. The assembler calculates the required value of the offset from the PC value of the B instruction to this label, then selects an encoding that sets `imm32` to that offset. Permitted offsets are even numbers in the range -1048576 to 1048574.  
 For encoding T4: the label of the instruction that is to be branched to. The assembler calculates the required value of the offset from the PC value of the B instruction to this label, then selects an encoding that sets `imm32` to that offset. Permitted offsets are even numbers in the range -16777216 to 16777214.

### Operation for all encodings

```
1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 BranchWritePC(PC + imm32);
```

## C2.4.16 BFC

Bit Field Clear. Bit Field Clear clears any number of adjacent bits at any position in a register, without affecting the other bits in the register.

### T1

Armv8-M Main Extension only

|    |    |    |    |    |     |   |   |   |   |   |   |   |   |   |   |    |      |    |    |    |      |     |   |   |   |   |   |   |   |   |     |
|----|----|----|----|----|-----|---|---|---|---|---|---|---|---|---|---|----|------|----|----|----|------|-----|---|---|---|---|---|---|---|---|-----|
| 15 | 14 | 13 | 12 | 11 | 10  | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | 15 | 14   | 13 | 12 | 11 | 10   | 9   | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0   |
| 1  | 1  | 1  | 1  | 0  | (0) | 1 | 1 | 0 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 0  | imm3 |    | Rd |    | imm2 | (0) |   |   |   |   |   |   |   |   | msb |

### T1 variant

BFC{<c>}{<q>} <Rd>, #<lsb>, #<width>

### Decode for this encoding

```

1 if !HaveMainExt() then UNDEFINED;
2 d = UInt(Rd); msbit = UInt(msb); lsbit = UInt(imm3:imm2);
3 if msbit < lsbit then UNPREDICTABLE;
4 if d IN {13,15} then UNPREDICTABLE;

```

### CONSTRAINED UNPREDICTABLE behavior

If `msbit < lsbit`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The value in the destination register is UNKNOWN.

### Assembler symbols for all encodings

|         |                                                                                                                 |
|---------|-----------------------------------------------------------------------------------------------------------------|
| <c>     | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                                        |
| <q>     | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                                        |
| <Rd>    | Is the general-purpose destination register, encoded in the "Rd" field.                                         |
| <lsb>   | Is the least significant bit that is to be cleared, in the range 0 to 31, encoded in the "imm3:imm2" field.     |
| <width> | Is the number of bits to be cleared, in the range 1 to 32-<lsb>, encoded in the "msb" field as <lsb>+<width>-1. |

### Operation for all encodings

```

1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 if msbit >= lsbit then
4 R[d][msbit:lsbit] = Replicate('0', msbit-lsbit+1);
5 // Other bits of R[d] are unchanged
6 else
7 R[d] = bits(32) UNKNOWN;

```

## C2.4.17 BFI

Bit Field Insert. Bit Field Insert copies any number of low order bits from a register into the same number of adjacent bits at any position in the destination register.

### T1

Armv8-M Main Extension only

|    |    |    |    |    |     |   |   |   |   |   |   |            |   |      |   |    |    |      |     |    |    |   |   |   |   |   |   |   |   |   |     |
|----|----|----|----|----|-----|---|---|---|---|---|---|------------|---|------|---|----|----|------|-----|----|----|---|---|---|---|---|---|---|---|---|-----|
| 15 | 14 | 13 | 12 | 11 | 10  | 9 | 8 | 7 | 6 | 5 | 4 | 3          | 2 | 1    | 0 | 15 | 14 | 13   | 12  | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0   |
| 1  | 1  | 1  | 1  | 0  | (0) | 1 | 1 | 0 | 1 | 1 | 0 | Rn != 1111 | 0 | imm3 |   | Rd |    | imm2 | (0) |    |    |   |   |   |   |   |   |   |   |   | msb |

### T1 variant

BFI{<c>}{<q>} <Rd>, <Rn>, #<lsb>, #<width>

### Decode for this encoding

```

1 if Rn == '1111' then SEE BFC;
2 if !HaveMainExt() then UNDEFINED;
3 d = UInt(Rd); n = UInt(Rn); msbit = UInt(msb); lsbit = UInt(imm3:imm2);
4 if msbit < lsbit then UNPREDICTABLE;
5 if d IN {13,15} || n == 13 then UNPREDICTABLE;

```

### CONSTRAINED UNPREDICTABLE behavior

If `msbit < lsbit`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The value in the destination register is UNKNOWN.

### Assembler symbols for all encodings

|         |                                                                                                                |
|---------|----------------------------------------------------------------------------------------------------------------|
| <c>     | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                                       |
| <q>     | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                                       |
| <Rd>    | Is the general-purpose destination register, encoded in the "Rd" field.                                        |
| <Rn>    | Is the general-purpose source register, encoded in the "Rn" field.                                             |
| <lsb>   | Is the least significant destination bit, in the range 0 to 31, encoded in the "imm3:imm2" field.              |
| <width> | Is the number of bits to be copied, in the range 1 to 32-<lsb>, encoded in the "msb" field as <lsb>+<width>-1. |

### Operation for all encodings

```

1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 if msbit >= lsbit then
4 R[d][msbit:lsbit] = R[n][(msbit-lsbit):0];
5 // Other bits of R[d] are unchanged
6 else
7 R[d] = bits(32) UNKNOWN;

```

## C2.4.18 BIC (immediate)

Bit Clear (immediate). Bit Clear (immediate) performs a bitwise AND of a register value and the complement of an immediate value, and writes the result to the destination register. It can optionally update the condition flags based on the result.

### T1

Armv8-M Main Extension only

|    |    |    |    |    |    |   |   |   |   |   |   |    |   |   |   |    |      |    |    |    |    |   |   |   |      |   |   |   |   |   |   |
|----|----|----|----|----|----|---|---|---|---|---|---|----|---|---|---|----|------|----|----|----|----|---|---|---|------|---|---|---|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3  | 2 | 1 | 0 | 15 | 14   | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6    | 5 | 4 | 3 | 2 | 1 | 0 |
| 1  | 1  | 1  | 1  | 0  | i  | 0 | 0 | 0 | 0 | 1 | S | Rn |   |   |   | 0  | imm3 |    |    |    | Rd |   |   |   | imm8 |   |   |   |   |   |   |

### BIC variant

Applies when **S** == 0.

BIC{<c>}{<q>} {<Rd>}, <Rn>, #<const>

### BICS variant

Applies when **S** == 1.

BICS{<c>}{<q>} {<Rd>}, <Rn>, #<const>

### Decode for this encoding

```

1 if !HaveMainExt() then UNDEFINED;
2 d = UInt(Rd); n = UInt(Rn); setflags = (S == '1');
3 (imm32, carry) = T32ExpandImm_C(i:imm3:imm8, APSR.C);
4 if d IN {13,15} || n IN {13,15} then UNPREDICTABLE;
```

### Assembler symbols for all encodings

|         |                                                                                                                                                                                                |
|---------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <c>     | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                                                                                                                       |
| <q>     | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                                                                                                                       |
| <Rd>    | Is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as <Rn>.                                                                         |
| <Rn>    | Is the general-purpose source register, encoded in the "Rn" field.                                                                                                                             |
| <const> | Is an immediate value derived from the 12-bit immediate that is encoded in the 'i:imm3:imm8' field. See <a href="#">C1.5 Modified immediate constants</a> on page 324 for the range of values. |

### Operation for all encodings

```

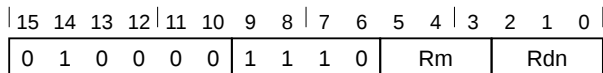
1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 result = R[n] AND NOT(imm32);
4 R[d] = result;
5 if setflags then
6 APSR.N = result[31];
7 APSR.Z = IsZeroBit(result);
8 APSR.C = carry;
9 // APSR.V unchanged
```

### C2.4.19 BIC (register)

Bit Clear (register). Bit Clear (register) performs a bitwise AND of a register value and the complement of an optionally-shifted register value, and writes the result to the destination register. It can optionally update the condition flags based on the result.

#### T1

Armv8-M



#### T1 variant

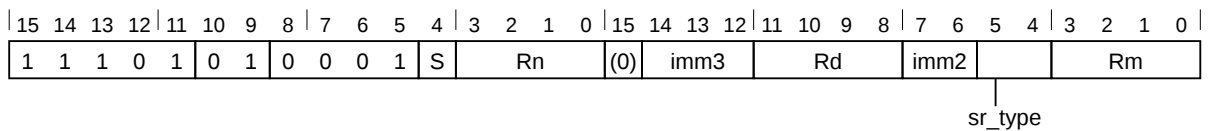
```
BIC<c>{<q>} {<Rdn>, } <Rdn>, <Rm>
 // Inside IT block
BICS{<q>} {<Rdn>, } <Rdn>, <Rm>
 // Outside IT block
```

#### Decode for this encoding

```
1 d = UInt(Rdn); n = UInt(Rdn); m = UInt(Rm); setflags = !InITBlock();
2 (shift_t, shift_n) = (SRTYPE_LSL, 0);
```

#### T2

Armv8-M Main Extension only



#### BIC, rotate right with extend variant

Applies when **S == 0** && **imm3 == 000** && **imm2 == 00** && **sr\_type == 11**.

```
BIC{<c>}{<q>} {<Rd>, } <Rn>, <Rm>, RRX
```

#### BIC, shift or rotate by value variant

Applies when **S == 0** && **!(imm3 == 000 && imm2 == 00 && sr\_type == 11)**.

```
BIC<c>.W {<Rd>, } <Rn>, <Rm>
 // Inside IT block, and <Rd>, <Rn>, <Rm> can be represented in T1
BIC{<c>}{<q>} {<Rd>, } <Rn>, <Rm> {, <shift> #<amount>}
```

#### BICS, rotate right with extend variant

Applies when **S == 1** && **imm3 == 000** && **imm2 == 00** && **sr\_type == 11**.

```
BICS{<c>}{<q>} {<Rd>, } <Rn>, <Rm>, RRX
```

#### BICS, shift or rotate by value variant

Applies when **S == 1** && **!(imm3 == 000 && imm2 == 00 && sr\_type == 11)**.

```
BICS.W {<Rd>, } <Rn>, <Rm>
 // Outside IT block, and <Rd>, <Rn>, <Rm> can be represented in T1
BICS{<c>}{<q>} {<Rd>, } <Rn>, <Rm> {, <shift> #<amount>}
```

### Decode for this encoding

```
1 if !HaveMainExt() then UNDEFINED;
2 d = UInt(Rd); n = UInt(Rn); m = UInt(Rm); setflags = (S == '1');
3 (shift_t, shift_n) = DecodeImmShift(sr_type, imm3:imm2);
4 if d IN {13,15} || n IN {13,15} || m IN {13,15} then UNPREDICTABLE;
```

### Assembler symbols for all encodings

<c> See [C1.2.5 Standard assembler syntax fields](#) on page 309.

<q> See [C1.2.5 Standard assembler syntax fields](#) on page 309.

<Rdn> Is the first general-purpose source register and the destination register, encoded in the "Rdn" field.

<Rd> Is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as <Rn>.

<Rn> Is the first general-purpose source register, encoded in the "Rn" field.

<Rm> Is the second general-purpose source register, encoded in the "Rm" field.

<shift> Is the type of shift to be applied to the second source register, encoded in the "sr\_type" field. It can have the following values:

- LSL when sr\_type = 00
- LSR when sr\_type = 01
- ASR when sr\_type = 10
- ROR when sr\_type = 11

<amount> Is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR) encoded in the "imm3:imm2" field as <amount> modulo 32.

### Operation for all encodings

```
1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 (shifted, carry) = Shift_C(R[m], shift_t, shift_n, APSR.C);
4 result = R[n] AND NOT(shifted);
5 R[d] = result;
6 if setflags then
7 APSR.N = result[31];
8 APSR.Z = IsZeroBit(result);
9 APSR.C = carry;
10 // APSR.V unchanged
```

## C2.4.20 BKPT

Breakpoint. Breakpoint causes a DebugMonitor exception or a debug halt to occur depending on the configuration of the debug support.

**BKPT** is an unconditional instruction and executes as such both inside and outside an **IT** instruction block.

### T1

Armv8-M

|    |    |    |    |    |    |   |   |      |   |   |   |   |   |   |   |
|----|----|----|----|----|----|---|---|------|---|---|---|---|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7    | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 1  | 0  | 1  | 1  | 1  | 1  | 1 | 0 | imm8 |   |   |   |   |   |   |   |

### T1 variant

BKPT{<q>} {#}<imm>

### Decode for this encoding

```
1 imm32 = ZeroExtend(imm8, 32);
2 // imm32 is for assembly/disassembly only and is ignored by hardware.
```

### Assembler symbols for all encodings

<q> See [C1.2.5 Standard assembler syntax fields](#) on page 309. A BKPT instruction must be unconditional.

<imm> Is an 8-bit unsigned immediate, in the range 0 to 255, encoded in the "imm8" field. The PE ignores this value, but a debugger might use it to store additional information about the breakpoint.

### Operation for all encodings

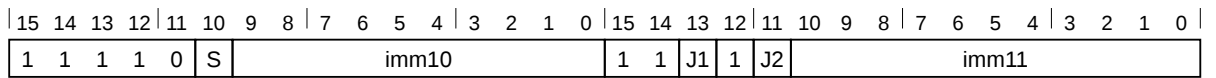
```
1 EncodingSpecificOperations();
2 BKPTInstrDebugEvent();
```

## C2.4.21 BL

Branch with Link (immediate). Branch with Link (immediate) calls a subroutine at a PC-relative address.

### T1

Armv8-M



### T1 variant

BL{<c>}{<q>} <label>

### Decode for this encoding

```
1 I1 = NOT(J1 EOR S); I2 = NOT(J2 EOR S); imm32 = SignExtend(S:I1:I2:imm10:imm11:'0', 32);
2 if InITBlock() && !LastInITBlock() then UNPREDICTABLE;
```

### Assembler symbols for all encodings

<c> See [C1.2.5 Standard assembler syntax fields](#) on page 309.  
 <q> See [C1.2.5 Standard assembler syntax fields](#) on page 309.  
 <label> The label of the instruction that is to be branched to. The assembler calculates the required value of the offset from the PC value of the BL instruction to this label, then selects an encoding with imm32 set to that offset. Permitted offsets are even numbers in the range -16777216 to 16777214.

### Operation for all encodings

```
1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 next_instr_addr = PC;
4 LR = next_instr_addr[31:1] : '1';
5 BranchWritePC(PC + imm32);
```



## C2.4.22 BLX, BLXNS

Branch with Link and Exchange (Non-secure). Branch with Link and Exchange calls a subroutine at an address, with the address and instruction set specified by a register. Bit[0] complies with the Arm architecture interworking rules for switching between the A32 and T32 instruction sets. However, Armv8-M only supports the T32 instruction set, so bit[0] must be 1. If bit[0] is 0 the PE takes an INVSTATE UsageFault exception on the instruction at the target address.

Branch with Link and Exchange Non-secure calls a subroutine at an address specified by a register, and if bit[0] of the target address is 0 then the instruction causes a transition from Secure to Non-secure state. This variant of the instruction must only be used when the additional steps required to make such a transition safe have been taken.

BLXNS is UNDEFINED if executed in Non-secure state, or if the Security Extension is not implemented.

### T1

Armv8-M

|    |    |    |    |    |    |   |   |   |   |   |    |    |     |     |   |
|----|----|----|----|----|----|---|---|---|---|---|----|----|-----|-----|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4  | 3  | 2   | 1   | 0 |
| 0  | 1  | 0  | 0  | 0  | 1  | 1 | 1 | 1 |   |   | Rm | NS | (0) | (0) |   |

#### BLX variant

Applies when **NS == 0**.

BLX{<c>}{<q>} <Rm>

#### BLXNS variant

Applies when **NS == 1**.

BLXNS{<c>}{<q>} <Rm>

#### Decode for this encoding

```

1 m = UInt(Rm); allowNonSecure = NS == '1';
2 if !IsSecure() && allowNonSecure then UNDEFINED;
3 if m IN {13,15} then UNPREDICTABLE;
4 if InITBlock() && !LastInITBlock() then UNPREDICTABLE;
```

#### Assembler symbols for all encodings

<c> See [C1.2.5 Standard assembler syntax fields](#) on page 309.  
 <q> See [C1.2.5 Standard assembler syntax fields](#) on page 309.  
 <Rm> Is the general-purpose register holding the address to be branched to, encoded in the "Rm" field. The SP can be used, but this is deprecated.

#### Operation for all encodings

```

1 if ConditionPassed() then
2 EncodingSpecificOperations();
3
4 target = R[m];
5 nextInstrAddr = PC - 2;
6 nextInstrAddr = nextInstrAddr[31:1] : '1';
7
8 if allowNonSecure && (target[0] == '0') then
9 if !IsAligned(SP, 8) then UNPREDICTABLE;
10 address = SP - 8;
11 RETPSR_Type savedPSR = Zeros();
12 savedPSR.Exception = IPSR.Exception;
13 savedPSR.SFPA = CONTROL_S.SFPA;
```

```

14 // Only the stack locations, not the store order, are architected
15 spName = LookUpSP();
16 mode = CurrentMode();
17 exc = Stack(address, 0, spName, mode, nextInstrAddr);
18 if exc.fault == NoFault then exc = Stack(address, 4, spName, mode, savedPSR);
19 HandleException(exc);
20 // Stack pointer update will raise a fault if limit violated
21 SP = address;
22 LR = 0xFEEEEFFF[31:0];
23 // If in handler mode, IPSR must be nonzero. To prevent revealing which
24 // Secure handler is calling Non-secure code, IPSR is set to an invalid but
25 // nonzero value (ie the reset exception number).
26 if mode == PEMode_Handler then
27 IPSR = 0x1[31:0];
28 else
29 LR = nextInstrAddr;
30
31 BLXWritePC(target, allowNonSecure);

```

### CONSTRAINED UNPREDICTABLE behavior

If `!IsAligned(SP, 8)`, then one of the following behaviors must occur:

- The instruction uses the current value of the stack pointer.
- The instruction behaves as though bits[2:0] of the stack pointer are 0b000.

### C2.4.23 BX, BXNS

Branch and Exchange (Non-secure). Branch and Exchange causes a branch to an address, with the address and instruction set specified by a register. Bit[0] complies with the Arm architecture interworking rules for switching between the A32 and T32 instruction sets. However, Armv8-M only supports the T32 instruction set, so bit[0] must be 1. If bit[0] is 0 the PE takes an INVSTATE UsageFault exception on the instruction at the target address.

Branch and Exchange Non-secure causes a branch to an address specified by a register. If bit[0] of the target address is 0, and the target address is not FNC\_RETURN or EXC\_RETURN, then the instruction causes a transition from Secure to Non-secure state. This variant of the instruction must only be used when the additional steps required to make such a transition safe have been taken.

BX can also be used for an exception return.

BXNS is UNDEFINED if executed in Non-secure state, or if the Security Extension is not implemented.

#### T1

Armv8-M

|    |    |    |    |    |    |   |   |   |    |   |   |    |     |     |   |
|----|----|----|----|----|----|---|---|---|----|---|---|----|-----|-----|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6  | 5 | 4 | 3  | 2   | 1   | 0 |
| 0  | 1  | 0  | 0  | 0  | 1  | 1 | 1 | 0 | Rm |   |   | NS | (0) | (0) |   |

#### BX variant

Applies when NS == 0.

BX{<c>}{<q>} <Rm>

#### BXNS variant

Applies when NS == 1.

BXNS{<c>}{<q>} <Rm>

#### Decode for this encoding

```

1 m = UInt(Rm); allowNonSecure = NS == '1';
2 if !IsSecure() && allowNonSecure then UNDEFINED;
3 if m IN {13,15} then UNPREDICTABLE;
4 if InITBlock() && !LastInITBlock() then UNPREDICTABLE;
```

#### Assembler symbols for all encodings

<c> See [C1.2.5 Standard assembler syntax fields](#) on page 309.  
 <q> See [C1.2.5 Standard assembler syntax fields](#) on page 309.  
 <Rm> Is the general-purpose register holding the address to be branched to, encoded in the "Rm" field. The SP can be used, but this is deprecated.

#### Operation for all encodings

```

1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 exc = BXWritePC(R[m], allowNonSecure);
4 HandleException(exc);
```

## C2.4.24 CBNZ, CBZ

Compare and Branch on Nonzero or Zero. Compare and Branch on Nonzero and Compare and Branch on Zero compare the value in a register with zero, and conditionally branch forward a constant value. They do not affect the condition flags.

### T1

Armv8-M

|    |    |    |    |    |    |   |   |      |   |   |   |   |    |   |   |
|----|----|----|----|----|----|---|---|------|---|---|---|---|----|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7    | 6 | 5 | 4 | 3 | 2  | 1 | 0 |
| 1  | 0  | 1  | 1  | op | 0  | i | 1 | imm5 |   |   |   |   | Rn |   |   |

#### CBNZ variant

Applies when `op == 1`.

CBNZ{<q>} <Rn>, <label>

#### CBZ variant

Applies when `op == 0`.

CBZ{<q>} <Rn>, <label>

#### Decode for this encoding

```
1 n = UInt(Rn); imm32 = ZeroExtend(i:imm5:'0', 32); nonzero = (op == '1');
2 if InITBlock() then UNPREDICTABLE;
```

#### Assembler symbols for all encodings

<q> See [C1.2.5 Standard assembler syntax fields](#) on page 309.  
<Rn> Is the general-purpose register to be tested, encoded in the "Rn" field.  
<label> Is the program label to be conditionally branched to. Its offset from the PC, a multiple of 2 in the range 0 to 126, is encoded as "i:imm5" times 4.

#### Operation for all encodings

```
1 EncodingSpecificOperations();
2 if nonzero != IsZero(R[n]) then
3 BranchWritePC(PC + imm32);
```

## C2.4.25 CDP, CDP2

Coprocessor Data Processing. Coprocessor Data Processing tells a coprocessor to perform an operation.

If no coprocessor can execute the instruction, a UsageFault exception is generated.

Arm reserves coprocessors CP8 to CP15, and this manual defines the valid instructions when coproc is in this range.

### T1

*Armv8-M Main Extension only*

|    |    |    |    |    |    |   |   |      |   |   |   |     |   |   |   |     |    |    |    |        |    |   |   |      |   |   |   |   |     |   |   |  |
|----|----|----|----|----|----|---|---|------|---|---|---|-----|---|---|---|-----|----|----|----|--------|----|---|---|------|---|---|---|---|-----|---|---|--|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7    | 6 | 5 | 4 | 3   | 2 | 1 | 0 | 15  | 14 | 13 | 12 | 11     | 10 | 9 | 8 | 7    | 6 | 5 | 4 | 3 | 2   | 1 | 0 |  |
| 1  | 1  | 1  | 0  | 1  | 1  | 1 | 0 | opc1 |   |   |   | CRn |   |   |   | CRd |    |    |    | coproc |    |   |   | opc2 |   |   |   | 0 | CRm |   |   |  |

#### T1 variant

CDP{<c>}{<q>} <coproc>, {#}<opc1>, <CRd>, <CRn>, <CRm> {, {#}<opc2>}

#### Decode for this encoding

```
1 if coproc IN '101x' then SEE "Floating-point";
2 if !HaveMainExt() then UNDEFINED;
3 cp = UInt(coproc);
```

### T2

*Armv8-M Main Extension only*

|    |    |    |    |    |    |   |   |      |   |   |   |     |   |   |   |     |    |    |    |        |    |   |   |      |   |   |   |   |     |   |   |  |
|----|----|----|----|----|----|---|---|------|---|---|---|-----|---|---|---|-----|----|----|----|--------|----|---|---|------|---|---|---|---|-----|---|---|--|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7    | 6 | 5 | 4 | 3   | 2 | 1 | 0 | 15  | 14 | 13 | 12 | 11     | 10 | 9 | 8 | 7    | 6 | 5 | 4 | 3 | 2   | 1 | 0 |  |
| 1  | 1  | 1  | 1  | 1  | 1  | 1 | 0 | opc1 |   |   |   | CRn |   |   |   | CRd |    |    |    | coproc |    |   |   | opc2 |   |   |   | 0 | CRm |   |   |  |

#### T2 variant

CDP2{<c>}{<q>} <coproc>, {#}<opc1>, <CRd>, <CRn>, <CRm> {, {#}<opc2>}

#### Decode for this encoding

```
1 if coproc IN '101x' then SEE "Floating-point";
2 if !HaveMainExt() then UNDEFINED;
3 cp = UInt(coproc);
```

#### Assembler symbols for all encodings

|          |                                                                                                             |
|----------|-------------------------------------------------------------------------------------------------------------|
| <c>      | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                                    |
| <q>      | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                                    |
| <coproc> | Is the name of the coprocessor, encoded in the "coproc" field. The generic coprocessor names are p0 to p15. |
| <opc1>   | Is a coprocessor-specific opcode, in the range 0 to 15, encoded in the "opc1" field.                        |
| <CRd>    | Is the destination coprocessor register, encoded in the "CRd" field.                                        |
| <CRn>    | Is the coprocessor register that contains the first operand, encoded in the "CRn" field.                    |
| <CRm>    | Is the coprocessor register that contains the second operand, encoded in the "CRm" field.                   |
| <opc2>   | Is a coprocessor-specific opcode in the range 0 to 7, defaulting to 0 and encoded in the "opc2" field.      |

#### Operation for all encodings

```
1 if ConditionPassed() then
```

```
2 EncodingSpecificOperations();
3 ExecuteCPCheck(cp);
4 if !Coprocc_Accepted(cp, ThisInstr()) then
5 GenerateCoproccException();
6 else
7 Coproc_InternalOperation(cp, ThisInstr());
```

## C2.4.26 CLREX

Clear Exclusive. Clear Exclusive clears the local record of the executing PE that an address has had a request for an exclusive access.

### T1

Armv8-M

|    |    |    |    |    |    |   |   |   |   |   |   |     |     |     |     |    |    |     |    |     |     |     |     |   |   |   |   |     |     |     |     |
|----|----|----|----|----|----|---|---|---|---|---|---|-----|-----|-----|-----|----|----|-----|----|-----|-----|-----|-----|---|---|---|---|-----|-----|-----|-----|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3   | 2   | 1   | 0   | 15 | 14 | 13  | 12 | 11  | 10  | 9   | 8   | 7 | 6 | 5 | 4 | 3   | 2   | 1   | 0   |
| 1  | 1  | 1  | 1  | 0  | 0  | 1 | 1 | 1 | 0 | 1 | 1 | (1) | (1) | (1) | (1) | 1  | 0  | (0) | 0  | (1) | (1) | (1) | (1) | 0 | 0 | 1 | 0 | (1) | (1) | (1) | (1) |

### T1 variant

CLREX{<c>}{<q>}

### Decode for this encoding

```
1 // No additional decoding required
```

### Assembler symbols for all encodings

<c> See [C1.2.5 Standard assembler syntax fields](#) on page 309.  
 <q> See [C1.2.5 Standard assembler syntax fields](#) on page 309.

### Operation for all encodings

```
1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 ClearExclusiveLocal(ProcessorID());
```

## C2.4.27 CLZ

Count Leading Zeros. Count Leading Zeros returns the number of binary zero bits before the first binary one bit in a value.

### T1

Armv8-M Main Extension only

|    |    |    |    |    |    |   |   |   |   |   |   |    |   |   |   |    |    |    |    |    |    |   |   |   |   |   |   |     |   |   |   |
|----|----|----|----|----|----|---|---|---|---|---|---|----|---|---|---|----|----|----|----|----|----|---|---|---|---|---|---|-----|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3  | 2 | 1 | 0 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3   | 2 | 1 | 0 |
| 1  | 1  | 1  | 1  | 1  | 0  | 1 | 0 | 1 | 0 | 1 | 1 | Rm |   |   |   | 1  | 1  | 1  | 1  | Rd |    |   |   | 1 | 0 | 0 | 0 | Rm2 |   |   |   |

### T1 variant

CLZ{<c>}{<q>} <Rd>, <Rm>

### Decode for this encoding

```

1 if !HaveMainExt() then UNDEFINED;
2 if Rm != Rm2 then UNPREDICTABLE;
3 d = UInt(Rd); m = UInt(Rm);
4 if d IN {13,15} || m IN {13,15} then UNPREDICTABLE;
```

### CONSTRAINED UNPREDICTABLE behavior

If  $Rm \neq Rm2$ , then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes as described, with no change to its behavior and no additional side effects.
- The value in the destination register is UNKNOWN.

### Assembler symbols for all encodings

<c> See [C1.2.5 Standard assembler syntax fields](#) on page 309.  
 <q> See [C1.2.5 Standard assembler syntax fields](#) on page 309.  
 <Rd> Is the general-purpose destination register, encoded in the "Rd" field.  
 <Rm> Is the general-purpose source register, encoded in the "Rm" field. Its number must be encoded twice.

### Operation for all encodings

```

1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 result = CountLeadingZeroBits(R[m]);
4 R[d] = result[31:0];
```



## C2.4.28 CMN (immediate)

Compare Negative (immediate). Compare Negative (immediate) adds a register value and an immediate value. It updates the condition flags based on the result, and discards the result.

### T1

Armv8-M Main Extension only

|    |    |    |    |    |    |   |   |   |   |   |   |    |   |   |   |    |      |    |    |    |    |   |   |   |      |   |   |   |   |   |   |  |
|----|----|----|----|----|----|---|---|---|---|---|---|----|---|---|---|----|------|----|----|----|----|---|---|---|------|---|---|---|---|---|---|--|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3  | 2 | 1 | 0 | 15 | 14   | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6    | 5 | 4 | 3 | 2 | 1 | 0 |  |
| 1  | 1  | 1  | 1  | 0  | i  | 0 | 1 | 0 | 0 | 0 | 1 | Rn |   |   |   | 0  | imm3 |    |    |    | 1  | 1 | 1 | 1 | imm8 |   |   |   |   |   |   |  |

### T1 variant

CMN{<c>}{<q>} <Rn>, #<const>

### Decode for this encoding

```

1 if !HaveMainExt() then UNDEFINED;
2 n = UInt(Rn); imm32 = T32ExpandImm(i:imm3:imm8);
3 if n == 15 then UNPREDICTABLE;
```

### Assembler symbols for all encodings

<c> See [C1.2.5 Standard assembler syntax fields](#) on page 309.  
 <q> See [C1.2.5 Standard assembler syntax fields](#) on page 309.  
 <Rn> Is the general-purpose source register, encoded in the "Rn" field.  
 <const> Is an immediate value derived from the 12-bit immediate that is encoded in the 'i:imm3:imm8' field. See [C1.5 Modified immediate constants](#) on page 324 for the range of values.

### Operation for all encodings

```

1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 (result, carry, overflow) = AddWithCarry(R[n], imm32, '0');
4 APSR.N = result[31];
5 APSR.Z = IsZeroBit(result);
6 APSR.C = carry;
7 APSR.V = overflow;
```



LSL when `sr_type = 00`  
 LSR when `sr_type = 01`  
 ASR when `sr_type = 10`  
 ROR when `sr_type = 11`  
 <amount> Is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR) encoded in the "imm3:imm2" field as <amount> modulo 32.

### Operation for all encodings

```

1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 shifted = Shift(R[m], shift_t, shift_n, APSR.C);
4 (result, carry, overflow) = AddWithCarry(R[n], shifted, '0');
5 APSR.N = result[31];
6 APSR.Z = IsZeroBit(result);
7 APSR.C = carry;
8 APSR.V = overflow;

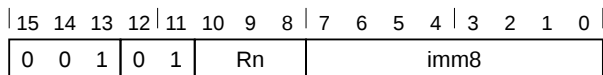
```

### C2.4.30 CMP (immediate)

Compare (immediate). Compare (immediate) subtracts an immediate value from a register value. It updates the condition flags based on the result, and discards the result.

#### T1

Armv8-M



#### T1 variant

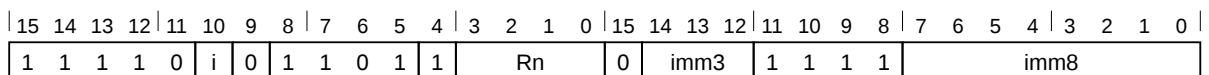
CMP {<c>} {<q>} <Rn>, #<imm8>

#### Decode for this encoding

```
1 n = UInt(Rn); imm32 = ZeroExtend(imm8, 32);
```

#### T2

Armv8-M Main Extension only



#### T2 variant

```
CMP {<c>}.W <Rn>, #<const>
// <Rn>, <const> can be represented in T1
CMP {<c>} {<q>} <Rn>, #<const>
```

#### Decode for this encoding

```
1 if !HaveMainExt() then UNDEFINED;
2 n = UInt(Rn); imm32 = T32ExpandImm(i:imm3:imm8);
3 if n == 15 then UNPREDICTABLE;
```

#### Assembler symbols for all encodings

|                            |                                                                                                                                                                                                |
|----------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>&lt;c&gt;</code>     | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                                                                                                                       |
| <code>&lt;q&gt;</code>     | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                                                                                                                       |
| <code>&lt;Rn&gt;</code>    | For encoding T1: is a general-purpose source register, encoded in the "Rn" field.<br>For encoding T2: is the general-purpose source register, encoded in the "Rn" field.                       |
| <code>&lt;imm8&gt;</code>  | Is an 8-bit unsigned immediate, in the range 0 to 255, encoded in the "imm8" field.                                                                                                            |
| <code>&lt;const&gt;</code> | Is an immediate value derived from the 12-bit immediate that is encoded in the 'i:imm3:imm8' field. See <a href="#">C1.5 Modified immediate constants</a> on page 324 for the range of values. |

#### Operation for all encodings

```
1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 (result, carry, overflow) = AddWithCarry(R[n], NOT(imm32), '1');
4 APSR.N = result[31];
5 APSR.Z = IsZeroBit(result);
6 APSR.C = carry;
7 APSR.V = overflow;
```



### Rotate right with extend variant

Applies when `imm3 == 000 && imm2 == 00 && sr_type == 11`.

```
CMP{<c>}{<q>} <Rn>, <Rm>, RRX
```

### Shift or rotate by value variant

Applies when `!(imm3 == 000 && imm2 == 00 && sr_type == 11)`.

```
CMP{<c>}.W <Rn>, <Rm>
 // <Rn>, <Rm> can be represented in T1 or T2
CMP{<c>}{<q>} <Rn>, <Rm>, <shift> #<amount>
```

### Decode for this encoding

```
1 if !HaveMainExt() then UNDEFINED;
2 n = UInt(Rn); m = UInt(Rm);
3 (shift_t, shift_n) = DecodeImmShift(sr_type, imm3:imm2);
4 if n == 15 || m IN {13,15} then UNPREDICTABLE;
```

### Assembler symbols for all encodings

|                             |                                                                                                                                                                                                                                                                                             |
|-----------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>&lt;c&gt;</code>      | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                                                                                                                                                                                                                    |
| <code>&lt;q&gt;</code>      | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                                                                                                                                                                                                                    |
| <code>&lt;Rn&gt;</code>     | For encoding T1 and T3: is the first general-purpose source register, encoded in the "Rn" field.<br>For encoding T2: is the first general-purpose source register, encoded in the "N:Rn" field.                                                                                             |
| <code>&lt;Rm&gt;</code>     | Is the second general-purpose source register, encoded in the "Rm" field.                                                                                                                                                                                                                   |
| <code>&lt;shift&gt;</code>  | Is the type of shift to be applied to the second source register, encoded in the "sr_type" field. It can have the following values:<br>LSL when <code>sr_type = 00</code><br>LSR when <code>sr_type = 01</code><br>ASR when <code>sr_type = 10</code><br>ROR when <code>sr_type = 11</code> |
| <code>&lt;amount&gt;</code> | Is the shift amount, in the range 1 to 31 (when <code>&lt;shift&gt; = LSL</code> or <code>ROR</code> ) or 1 to 32 (when <code>&lt;shift&gt; = LSR</code> or <code>ASR</code> ) encoded in the "imm3:imm2" field as <code>&lt;amount&gt;</code> modulo 32.                                   |

### Operation for all encodings

```
1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 shifted = Shift(R[m], shift_t, shift_n, APSR.C);
4 (result, carry, overflow) = AddWithCarry(R[n], NOT(shifted), '1');
5 APSR.N = result[31];
6 APSR.Z = IsZeroBit(result);
7 APSR.C = carry;
8 APSR.V = overflow;
```

### C2.4.32 CPS

Change PE State. Change PE State. The instruction modifies the PRIMASK and FAULTMASK special-purpose register values.

#### T1

Armv8-M

|    |    |    |    |    |    |   |   |   |   |   |    |     |     |   |   |
|----|----|----|----|----|----|---|---|---|---|---|----|-----|-----|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4  | 3   | 2   | 1 | 0 |
| 1  | 0  | 1  | 1  | 0  | 1  | 1 | 0 | 0 | 1 | 1 | im | (0) | (0) | I | F |

#### CPSID variant

Applies when `im == 1`.

CPSID{<q>} <iflags>

#### CPSIE variant

Applies when `im == 0`.

CPSIE{<q>} <iflags>

#### Decode for this encoding

```

1 enable = (im == '0'); disable = (im == '1');
2 if InITBlock() then UNPREDICTABLE;
3 if (I == '0' && F == '0') then UNPREDICTABLE;
4 affectPRI = (I == '1'); affectFAULT = (F == '1');
5 if !HaveMainExt() then
6 if (I == '0') then UNPREDICTABLE;
7 if (F == '1') then UNPREDICTABLE;
```

#### CONSTRAINED UNPREDICTABLE behavior

If `I == '0' && F == '0'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.

#### CONSTRAINED UNPREDICTABLE behavior

If `!HaveMainExt() && (I == '0' || F == '1')`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.

#### Assembler symbols for all encodings

- <q> See [C1.2.5 Standard assembler syntax fields](#) on page 309.
- <iflags> Is a sequence of one or more of the following, specifying which interrupt mask bits are affected:
- f FAULTMASK. When set to 1, raises the execution priority to -1, the same priority as `HardFault`. This is a 1-bit register, that can be updated only by privileged software. The register clears to 0 on return from any exception other than `NMI`.
  - i PRIMASK. When set to 1, raises the execution priority to 0. This is a 1-bit register, that can be updated only by privileged software.

### Operation for all encodings

```
1 EncodingSpecificOperations();
2 if CurrentModeIsPrivileged() then
3 if enable then
4 if affectPRI then
5 PRIMASK.PM = '0';
6 if affectFAULT then
7 FAULTMASK.FM = '0';
8 if disable then
9 if affectPRI then
10 PRIMASK.PM = '1';
11 if affectFAULT && ExecutionPriority() > -1 then
12 FAULTMASK.FM = '1';
```



### C2.4.33 CSDB

Consumption of Speculative Data Barrier. Consumption of Speculative Data Barrier is a memory barrier that controls speculative execution and data value prediction.

No instruction other than branch instructions and instructions that write to the PC appearing in program order after the CSDB can be speculatively executed using the results of any:

- Data value predictions of any instructions.
- APSR.{N,Z,C,V} predictions of any instructions other than conditional branch instructions and conditional instructions that write to the PC appearing in program order before the CSDB that have not been architecturally resolved.

APSR.{N,Z,C,V} is not considered a data value. This instruction permits:

- Control flow speculation before and after the CSDB.
- Speculative execution of conditional data processing instructions after the CSDB, unless they use the results of data value or APSR.{N,Z,C,V} predictions of instructions appearing in program order before the CSDB that have not been architecturally resolved.

#### T1

Armv8-M Main Extension only

|    |    |    |    |    |    |   |   |   |   |   |   |     |     |     |     |    |    |     |    |     |    |   |   |   |   |   |   |   |   |   |   |   |
|----|----|----|----|----|----|---|---|---|---|---|---|-----|-----|-----|-----|----|----|-----|----|-----|----|---|---|---|---|---|---|---|---|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3   | 2   | 1   | 0   | 15 | 14 | 13  | 12 | 11  | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |   |
| 1  | 1  | 1  | 1  | 0  | 0  | 1 | 1 | 1 | 0 | 1 | 0 | (1) | (1) | (1) | (1) | 1  | 0  | (0) | 0  | (0) | 0  | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 |

#### T1 variant

CSDB{<c>}.W

#### Decode for this encoding

```
1 if !HaveMainExt() then UNDEFINED;
2 if InITBlock() then UNPREDICTABLE;
```

#### CONSTRAINED UNPREDICTABLE behavior

If InITBlock(), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes unconditionally.
- The instruction executes conditionally.

#### Assembler symbols for all encodings

<c> See [C1.2.5 Standard assembler syntax fields](#) on page 309.

#### Operation for all encodings

```
1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 ConsumptionOfSpeculativeDataBarrier();
```

### C2.4.34 DBG

Debug hint. Debug Hint provides a hint to debug trace support and related debug systems. See debug architecture documentation for what use (if any) is made of this instruction.

DBG is a NOP-compatible hint.

#### T1

*Armv8-M Main Extension only*

|    |    |    |    |    |    |   |   |   |   |   |   |     |     |     |     |    |    |     |    |     |    |   |   |   |   |   |   |        |   |   |   |
|----|----|----|----|----|----|---|---|---|---|---|---|-----|-----|-----|-----|----|----|-----|----|-----|----|---|---|---|---|---|---|--------|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3   | 2   | 1   | 0   | 15 | 14 | 13  | 12 | 11  | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3      | 2 | 1 | 0 |
| 1  | 1  | 1  | 1  | 0  | 0  | 1 | 1 | 1 | 0 | 1 | 0 | (1) | (1) | (1) | (1) | 1  | 0  | (0) | 0  | (0) | 0  | 0 | 0 | 1 | 1 | 1 | 1 | option |   |   |   |

#### T1 variant

DBG{<c>}{<q>} #<option>

#### Decode for this encoding

```
1 if !HaveMainExt() then UNDEFINED;
2 // Any decoding of 'option' is specified by the debug system
```

#### Assembler symbols for all encodings

<c> See [C1.2.5 Standard assembler syntax fields](#) on page 309.  
 <q> See [C1.2.5 Standard assembler syntax fields](#) on page 309.  
 <option> Is a 4-bit unsigned immediate, in the range 0 to 15, encoded in the "option" field.

#### Operation for all encodings

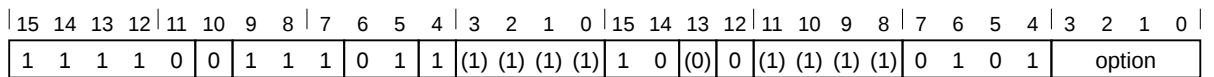
```
1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 Hint_Debug(option);
```

### C2.4.35 DMB

Data Memory Barrier. Data Memory Barrier acts as a memory barrier. It ensures that all explicit memory accesses that appear in program order before the DMB instruction are observed before any explicit memory accesses that appear in program order after the DMB instruction. It does not affect the ordering of any other instructions executing on the PE.

#### T1

Armv8-M



#### T1 variant

DMB{<c>}{<q>} {<option>}

#### Decode for this encoding

```
1 // No additional decoding required
```

#### Assembler symbols for all encodings

- <c> See [C1.2.5 Standard assembler syntax fields](#) on page 309.
- <q> See [C1.2.5 Standard assembler syntax fields](#) on page 309.
- <option> Specifies an optional limitation on the barrier operation. Values are:
  - SY Full system barrier operation, encoded as option = 0b11111. Can be omitted.
 All other encodings of option are reserved. The corresponding instructions execute as full system barrier operations, but must not be relied upon by software.

#### Operation for all encodings

```
1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 DataMemoryBarrier(option);
```

### C2.4.36 DSB

Data Synchronization Barrier. Data Synchronization Barrier acts as a special kind of memory barrier. No instruction in program order after this instruction can execute until this instruction completes. This instruction completes only when both:

- Any explicit memory access made before this instruction is complete.
- The side-effects of any SCS access that performs a context-altering operation are visible.

#### T1

Armv8-M

|    |    |    |    |    |    |   |   |   |   |   |   |     |     |     |     |    |    |     |    |     |     |     |     |   |   |   |   |        |   |   |   |
|----|----|----|----|----|----|---|---|---|---|---|---|-----|-----|-----|-----|----|----|-----|----|-----|-----|-----|-----|---|---|---|---|--------|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3   | 2   | 1   | 0   | 15 | 14 | 13  | 12 | 11  | 10  | 9   | 8   | 7 | 6 | 5 | 4 | 3      | 2 | 1 | 0 |
| 1  | 1  | 1  | 1  | 0  | 0  | 1 | 1 | 1 | 0 | 1 | 1 | (1) | (1) | (1) | (1) | 1  | 0  | (0) | 0  | (1) | (1) | (1) | (1) | 0 | 1 | 0 | 0 | option |   |   |   |

#### T1 variant

DSB{<c>}{<q>} {<option>}

#### Decode for this encoding

```
1 // No additional decoding required
```

#### Assembler symbols for all encodings

<c> See [C1.2.5 Standard assembler syntax fields](#) on page 309.  
 <q> See [C1.2.5 Standard assembler syntax fields](#) on page 309.  
 <option> Specifies an optional limitation on the barrier operation. Values are:  
 SY Full system barrier operation, encoded as option = 0b11111. Can be omitted.  
 All other encodings of option are reserved. The corresponding instructions execute as full system barrier operations, but must not be relied upon by software.

#### Operation for all encodings

```
1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 DataSynchronizationBarrier(option);
```

### C2.4.37 EOR (immediate)

Exclusive OR (immediate). Exclusive OR (immediate) performs a bitwise Exclusive OR of a register value and an immediate value, and writes the result to the destination register. It can optionally update the condition flags based on the result.

#### T1

Armv8-M Main Extension only

|    |    |    |    |    |    |   |   |   |   |   |   |    |   |   |   |    |      |    |    |    |    |   |   |   |      |   |   |   |   |   |   |
|----|----|----|----|----|----|---|---|---|---|---|---|----|---|---|---|----|------|----|----|----|----|---|---|---|------|---|---|---|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3  | 2 | 1 | 0 | 15 | 14   | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6    | 5 | 4 | 3 | 2 | 1 | 0 |
| 1  | 1  | 1  | 1  | 0  | i  | 0 | 0 | 1 | 0 | 0 | S | Rn |   |   |   | 0  | imm3 |    |    |    | Rd |   |   |   | imm8 |   |   |   |   |   |   |

#### EOR variant

Applies when **S == 0**.

EOR{<c>}{<q>} {<Rd>}, <Rn>, #<const>

#### EORS variant

Applies when **S == 1 && Rd != 1111**.

EORS{<c>}{<q>} {<Rd>}, <Rn>, #<const>

#### Decode for this encoding

```

1 if Rd == '1111' && S == '1' then SEE "TEQ (immediate)";
2 if !HaveMainExt() then UNDEFINED;
3 d = UInt(Rd); n = UInt(Rn); setflags = (S == '1');
4 (imm32, carry) = T32ExpandImm_C(i:imm3:imm8, APSR.C);
5 if d == 13 || (d == 15 && S == '0') || n IN {13,15} then UNPREDICTABLE;

```

#### Assembler symbols for all encodings

<c> See [C1.2.5 Standard assembler syntax fields](#) on page 309.  
 <q> See [C1.2.5 Standard assembler syntax fields](#) on page 309.  
 <Rd> Is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as <Rn>.  
 <Rn> Is the general-purpose source register, encoded in the "Rn" field.  
 <const> Is an immediate value derived from the 12-bit immediate that is encoded in the 'i:imm3:imm8' field. See [C1.5 Modified immediate constants](#) on page 324 for the range of values.

#### Operation for all encodings

```

1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 result = R[n] EOR imm32;
4 R[d] = result;
5 if setflags then
6 APSR.N = result[31];
7 APSR.Z = IsZeroBit(result);
8 APSR.C = carry;
9 // APSR.V unchanged

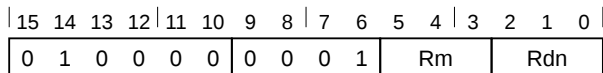
```

### C2.4.38 EOR (register)

Exclusive OR (register). Exclusive OR (register) performs a bitwise Exclusive OR of a register value and an optionally-shifted register value, and writes the result to the destination register. It can optionally update the condition flags based on the result.

#### T1

Armv8-M



#### T1 variant

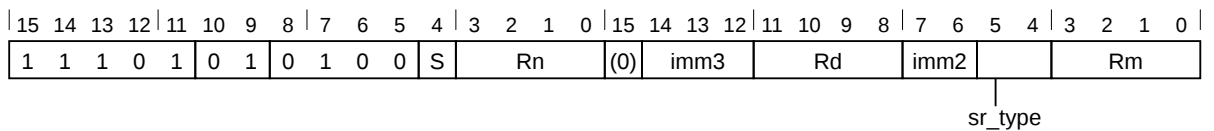
```
EOR<c>{<q>} {<Rdn>, } <Rdn>, <Rm>
 // Inside IT block
EORS{<q>} {<Rdn>, } <Rdn>, <Rm>
 // Outside IT block
```

#### Decode for this encoding

```
1 d = UInt(Rdn); n = UInt(Rdn); m = UInt(Rm); setflags = !InITBlock();
2 (shift_t, shift_n) = (SRTYPE_LSL, 0);
```

#### T2

Armv8-M Main Extension only



#### EOR, rotate right with extend variant

Applies when **S == 0** && **imm3 == 000** && **imm2 == 00** && **sr\_type == 11**.

```
EOR{<c>}{<q>} {<Rd>, } <Rn>, <Rm>, RRX
```

#### EOR, shift or rotate by value variant

Applies when **S == 0** && **!(imm3 == 000 && imm2 == 00 && sr\_type == 11)**.

```
EOR<c>.W {<Rd>, } <Rn>, <Rm>
 // Inside IT block, and <Rd>, <Rn>, <Rm> can be represented in T1
EOR{<c>}{<q>} {<Rd>, } <Rn>, <Rm> {, <shift> #<amount>}
```

#### EORS, rotate right with extend variant

Applies when **S == 1** && **imm3 == 000** && **Rd != 1111** && **imm2 == 00** && **sr\_type == 11**.

```
EORS{<c>}{<q>} {<Rd>, } <Rn>, <Rm>, RRX
```

#### EORS, shift or rotate by value variant

Applies when **S == 1** && **!(imm3 == 000 && imm2 == 00 && sr\_type == 11) && Rd != 1111**.

```
EORS.W {<Rd>,} <Rn>, <Rm>
// Outside IT block, and <Rd>, <Rn>, <Rm> can be represented in T1
EORS{<c>}{<q>} {<Rd>,} <Rn>, <Rm> {, <shift> #<amount>}
```

### Decode for this encoding

```
1 if Rd == '1111' && S == '1' then SEE "TEQ (register)";
2 if !HaveMainExt() then UNDEFINED;
3 d = UInt(Rd); n = UInt(Rn); m = UInt(Rm); setflags = (S == '1');
4 (shift_t, shift_n) = DecodeImmShift(sr_type, imm3:imm2);
5 if d == 13 || (d == 15 && S == '0') || n IN {13,15} || m IN {13,15} then UNPREDICTABLE;
```

### Assembler symbols for all encodings

<c> See [C1.2.5 Standard assembler syntax fields](#) on page 309.

<q> See [C1.2.5 Standard assembler syntax fields](#) on page 309.

<Rdn> Is the first general-purpose source register and the destination register, encoded in the "Rdn" field.

<Rd> Is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as <Rn>.

<Rn> Is the first general-purpose source register, encoded in the "Rn" field.

<Rm> Is the second general-purpose source register, encoded in the "Rm" field.

<shift> Is the type of shift to be applied to the second source register, encoded in the "sr\_type" field. It can have the following values:

|     |                   |
|-----|-------------------|
| LSL | when sr_type = 00 |
| LSR | when sr_type = 01 |
| ASR | when sr_type = 10 |
| ROR | when sr_type = 11 |

<amount> Is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR) encoded in the "imm3:imm2" field as <amount> modulo 32.

### Operation for all encodings

```
1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 (shifted, carry) = Shift_C(R[m], shift_t, shift_n, APSR.C);
4 result = R[n] EOR shifted;
5 R[d] = result;
6 if setflags then
7 APSR.N = result[31];
8 APSR.Z = IsZeroBit(result);
9 APSR.C = carry;
10 // APSR.V unchanged
```

### C2.4.39 FLDMDBX, FLDMIAX

FLDMX (Decrement Before, Increment After). FLDMX (Decrement Before, Increment After) loads multiple extension registers from consecutive memory locations using an address from a general-purpose register.

Arm deprecates use of FLDMDBX and FLDMIAX, except for disassembly purposes, and reassembly of disassembled code.

#### T1

Armv8-M Floating-point Extension only

|    |    |    |    |    |    |   |   |   |   |   |   |    |   |   |   |    |    |    |    |    |    |   |   |             |   |   |   |   |   |   |   |
|----|----|----|----|----|----|---|---|---|---|---|---|----|---|---|---|----|----|----|----|----|----|---|---|-------------|---|---|---|---|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3  | 2 | 1 | 0 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7           | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 1  | 1  | 1  | 0  | 1  | 1  | 0 | P | U | D | W | 1 | Rn |   |   |   | Vd |    |    |    | 1  | 0  | 1 | 1 | imm8<0> = 1 |   |   |   |   |   |   |   |

#### Decrement Before variant

Applies when **P == 1 && U == 0 && W == 1**.

FLDMDBX{<c>}{<q>} <Rn>{!}, <dreglist>

#### Increment After variant

Applies when **P == 0 && U == 1**.

FLDMIAX{<c>}{<q>} <Rn>{!}, <dreglist>

#### Decode for this encoding

```

1 if P == '0' && U == '0' && W == '0' then SEE "Related encodings";
2 if P == '1' && W == '0' then SEE VLDR;
3 CheckDecodeFaults();
4 if P == U && W == '1' then UNDEFINED;
5 // Remaining combinations are PUW = 010 (IA without !), 011 (IA with !), 101 (DB with !)
6 single_regs = FALSE; add = (U == '1'); wback = (W == '1');
7 d = UInt(D:Vd); n = UInt(Rn); imm32 = ZeroExtend(imm8:'00', 32);
8 regs = UInt(imm8) DIV 2;
9 if n == 15 then UNPREDICTABLE;
10 if regs == 0 || regs > 16 || (d+regs) > 32 then UNPREDICTABLE;
11 if VFPSmallRegisterBank() && (d+regs) > 16 then UNPREDICTABLE;
```

#### CONSTRAINED UNPREDICTABLE behavior

If `regs == 0`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction operates as a FLDMX with the same addressing mode but loads no registers.

#### CONSTRAINED UNPREDICTABLE behavior

If `regs > 16 || (d+regs) > 32`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- One or more of the floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.



### CONSTRAINED UNPREDICTABLE behavior

If `VFP_SmallRegisterBank() && (d+regs) > 16`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.

### Assembler symbols for all encodings

|            |                                                                                                                                                                                                                                                                                                                                                             |
|------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <c>        | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                                                                                                                                                                                                                                                                                    |
| <q>        | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                                                                                                                                                                                                                                                                                    |
| <Rn>       | Is the general-purpose base register, encoded in the "Rn" field.                                                                                                                                                                                                                                                                                            |
| !          | Specifies base register writeback. Encoded in the "W" field as 1 if present, otherwise 0.                                                                                                                                                                                                                                                                   |
| <dreglist> | Is the list of consecutively numbered 64-bit floating-point registers to be transferred. The first register in the list is encoded in "D:Vd", and "imm8" is set to twice the number of registers in the list plus one. The list must contain at least one register, all registers must be in the range D0-D15, and must not contain more than 16 registers. |

### Operation for all encodings

```

1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 ExecuteFPCheck();
4 address = if add then R[n] else R[n]-imm32;
5 regval = if add then R[n]+imm32 else R[n]-imm32;
6
7 // Determine if the stack pointer limit must be checked
8 if n == 13 && wback then
9 (limit, applylimit) = LookUpSPLim(LookUpSP());
10 // If memory operation is not performed as a result of a stack limit violation,
11 // and the write-back of the SP itself does not raise a stack limit violation, it
12 // is "IMPLEMENTATION_DEFINED" whether a SPLIM exception is raised.
13 // Arm recommends that any instruction which discards a memory access as
14 // a result of a stack limit violation, and where the write-back of the SP itself
15 // does not raise a stack limit violation, generates an SPLIM exception.
16 if boolean IMPLEMENTATION_DEFINED "SPLIM exception on invalid memory access" then
17 if applylimit && (UInt(address) < UInt(limit)) then
18 if HaveMainExt() then
19 UFSR.STKOF = '1';
20 // If Main Extension is not implemented the fault always escalates to
21 // HardFault
22 excInfo = CreateException(UsageFault, FALSE, boolean UNKNOWN);
23 HandleException(excInfo);
24 else
25 applylimit = FALSE;
26
27 // Memory operation only performed if limit not violated
28 if !applylimit || (UInt(regval) >= UInt(limit)) then
29 for r = 0 to regs-1
30 if single_regs then
31 S[d+r] = MemA[address,4];
32 address = address+4;
33 else
34 word1 = MemA[address,4]; word2 = MemA[address+4,4];
35 address = address+8;
36 // Combine the word-aligned words in the correct order for
37 // current endianness.
38 D[d+r] = if BigEndian() then word1:word2 else word2:word1;
39
40 // If the stack pointer is being updated a fault will be raised if
41 // the limit is violated
42 if wback then RSPCheck[n] = regval;

```

## C2.4.40 FSTMDBX, FSTMIAX

FSTMX (Decrement Before, Increment After). FSTMX (Decrement Before, Increment After) stores multiple extension registers to consecutive memory locations using an address from a general-purpose register.

Arm deprecates use of FSTMDBX and FSTMIAX, except for disassembly purposes, and reassembly of disassembled code.

### T1

Armv8-M Floating-point Extension only

|    |    |    |    |    |    |   |   |   |   |   |   |    |   |   |   |    |    |    |    |    |    |   |   |             |   |   |   |   |   |   |   |
|----|----|----|----|----|----|---|---|---|---|---|---|----|---|---|---|----|----|----|----|----|----|---|---|-------------|---|---|---|---|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3  | 2 | 1 | 0 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7           | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 1  | 1  | 1  | 0  | 1  | 1  | 0 | P | U | D | W | 0 | Rn |   |   |   | Vd |    |    |    | 1  | 0  | 1 | 1 | imm8<0> = 1 |   |   |   |   |   |   |   |

#### Decrement Before variant

Applies when **P == 1 && U == 0 && W == 1**.

FSTMDBX{<c>}{<q>} <Rn>{!}, <dreglist>

#### Increment After variant

Applies when **P == 0 && U == 1**.

FSTMIAX{<c>}{<q>} <Rn>{!}, <dreglist>

#### Decode for this encoding

```

1 if P == '0' && U == '0' && W == '0' then SEE "Related encodings";
2 if P == '1' && W == '0' then SEE VSTR;
3 CheckDecodeFaults();
4 if P == U && W == '1' then UNDEFINED;
5 // Remaining combinations are PUW = 010 (IA without !), 011 (IA with !), 101 (DB with !)
6 single_regs = FALSE; add = (U == '1'); wback = (W == '1');
7 d = UInt(D:Vd); n = UInt(Rn); imm32 = ZeroExtend(imm8:'00', 32);
8 regs = UInt(imm8) DIV 2;
9 if n == 15 then UNPREDICTABLE;
10 if regs == 0 || regs > 16 || (d+regs) > 32 then UNPREDICTABLE;
11 if VFPSmallRegisterBank() && (d+regs) > 16 then UNPREDICTABLE;
```

#### CONSTRAINED UNPREDICTABLE behavior

If `regs == 0`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction operates as a FSTMX with the same addressing mode but stores no registers.

#### CONSTRAINED UNPREDICTABLE behavior

If `regs > 16 || (d+regs) > 32`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The memory locations specified by the instruction and the number of registers specified by the instruction if the register list had not gone out of range, become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

### CONSTRAINED UNPREDICTABLE behavior

If `VFPSmallRegisterBank() && (d+regs) > 16`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.

### Assembler symbols for all encodings

|            |                                                                                                                                                                                                                                                                                                                                                             |
|------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <c>        | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                                                                                                                                                                                                                                                                                    |
| <q>        | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                                                                                                                                                                                                                                                                                    |
| <Rn>       | Is the general-purpose base register, encoded in the "Rn" field.                                                                                                                                                                                                                                                                                            |
| !          | Specifies base register writeback. Encoded in the "W" field as 1 if present, otherwise 0.                                                                                                                                                                                                                                                                   |
| <dreglist> | Is the list of consecutively numbered 64-bit floating-point registers to be transferred. The first register in the list is encoded in "D:Vd", and "imm8" is set to twice the number of registers in the list plus one. The list must contain at least one register, all registers must be in the range D0-D15, and must not contain more than 16 registers. |

### Operation for all encodings

```
1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 ExecuteFPCheck();
4 address = if add then R[n] else R[n]-imm32;
5 regval = if add then R[n]+imm32 else R[n]-imm32;
6
7 // Determine if the stack pointer limit should be checked
8 if n == 13 && wback then
9 (limit, applylimit) = LookUpSPLim(LookUpSP());
10 else
11 applylimit = FALSE;
12
13 // Memory operation only performed if limit not violated
14 if !applylimit || (UInt(regval) >= UInt(limit)) then
15 for r = 0 to regs-1
16 if single_regs then
17 MemA[address,4] = S[d+r];
18 address = address+4;
19 else
20 // Store as two word-aligned words in the correct order for current
21 // endianness.
22 MemA[address,4] = if BigEndian() then D[d+r][63:32] else D[d+r][31:0];
23 MemA[address+4,4] = if BigEndian() then D[d+r][31:0] else D[d+r][63:32];
24 address = address+8;
25
26 // If the stack pointer is being updated a fault will be raised if
27 // the limit is violated
28 if wback then RSPCheck[n] = regval;
```

## C2.4.41 ISB

Instruction Synchronization Barrier. Instruction Synchronization Barrier flushes the pipeline in the PE and is a context synchronization event.

### T1

Armv8-M

|    |    |    |    |    |    |   |   |   |   |   |   |     |     |     |     |    |    |     |    |     |     |     |     |   |   |   |   |        |   |   |   |
|----|----|----|----|----|----|---|---|---|---|---|---|-----|-----|-----|-----|----|----|-----|----|-----|-----|-----|-----|---|---|---|---|--------|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3   | 2   | 1   | 0   | 15 | 14 | 13  | 12 | 11  | 10  | 9   | 8   | 7 | 6 | 5 | 4 | 3      | 2 | 1 | 0 |
| 1  | 1  | 1  | 1  | 0  | 0  | 1 | 1 | 1 | 0 | 1 | 1 | (1) | (1) | (1) | (1) | 1  | 0  | (0) | 0  | (1) | (1) | (1) | (1) | 0 | 1 | 1 | 0 | option |   |   |   |

### T1 variant

ISB{<c>}{<q>} {<option>}

### Decode for this encoding

```
1 // No additional decoding required
```

### Assembler symbols for all encodings

<c> See [C1.2.5 Standard assembler syntax fields](#) on page 309.  
 <q> See [C1.2.5 Standard assembler syntax fields](#) on page 309.  
 <option> Specifies an optional limitation on the barrier operation. Values are:  
     SY Full system barrier operation, encoded as option = 0b11111. Can be omitted.  
 All other encodings of option are reserved. The corresponding instructions execute as full system barrier operations, but must not be relied upon by software.

### Operation for all encodings

```
1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 InstructionSynchronizationBarrier(option);
```

## C2.4.42 IT

If-Then. If Then makes up to four following instructions (the IT block) conditional. The conditions for the instructions in the IT block can be the same, or some of them can be the inverse of others.

IT does not affect the condition code flags. Branches to any instruction in the IT block are not permitted, apart from those performed by exception returns.

16-bit instructions in the IT block, other than [CMP \(register\)](#), [CMN \(register\)](#), and [TST \(register\)](#), do not set the condition code flags. The AL condition can be specified to get this changed behavior without conditional execution.

### T1

Armv8-M Main Extension only

|    |    |    |    |    |    |   |   |           |   |   |   |              |   |   |   |
|----|----|----|----|----|----|---|---|-----------|---|---|---|--------------|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7         | 6 | 5 | 4 | 3            | 2 | 1 | 0 |
| 1  | 0  | 1  | 1  | 1  | 1  | 1 | 1 | firstcond |   |   |   | mask != 0000 |   |   |   |

### T1 variant

IT{<x>{<y>{<z>}}}{<q>} <cond>

### Decode for this encoding

```

1 if mask == '0000' then SEE "Related encodings";
2 if !HaveMainExt() then UNDEFINED;
3 if firstcond == '1111' || (firstcond == '1110' && BitCount(mask) != 1) then UNPREDICTABLE;
4 if InITBlock() then UNPREDICTABLE;

```

### CONSTRAINED UNPREDICTABLE behavior

If `firstcond == '1111' || (firstcond == '1110' && BitCount(mask) != 1)`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The '1111' condition is treated as being the same as the '1110' condition, meaning always, and the ITSTATE state machine is progressed in the same way as for any other `cond_base` value.

### Assembler symbols for all encodings

|        |                                                                                                                                                                                                                                                                      |
|--------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <x>    | The condition for the second instruction in the IT block. If omitted, the "mask" field is set to 0b1000. If present it is encoded in the "mask[3]" field:<br>E NOT firstcond[0]<br>T firstcond[0]                                                                    |
| <y>    | The condition for the third instruction in the IT block. If omitted and <x> is present, the "mask[2:0]" field is set to 0b100. If <y> is present it is encoded in the "mask[2]" field:<br>E NOT firstcond[0]<br>T firstcond[0]                                       |
| <z>    | The condition for the fourth instruction in the IT block. If omitted and <y> is present, the "mask[1:0]" field is set to 0b10. If <z> is present, the "mask[0]" field is set to 1, and it is encoded in the "mask[1]" field:<br>E NOT firstcond[0]<br>T firstcond[0] |
| <q>    | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                                                                                                                                                                                             |
| <cond> | The condition for the first instruction in the IT block, encoded in the "firstcond" field. See <a href="#">C1.3 Conditional execution</a> on page 313 for the range of conditions available, and the encodings.                                                      |

### Operation for all encodings

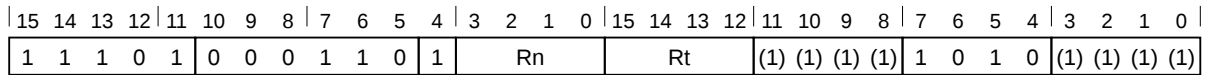
```
1 EncodingSpecificOperations();
2 ITSTATE[7:0] = firstcond:mask;
```

### C2.4.43 LDA

Load-Acquire Word. Load-Acquire Word loads a word from memory and writes it to a register. The instruction also has memory ordering semantics.

#### T1

Armv8-M



#### T1 variant

LDA{<c>}{<q>} <Rt>, [<Rn>]

#### Decode for this encoding

```
1 t = UInt(Rt); n = UInt(Rn);
2 if t IN {13,15} || n == 15 then UNPREDICTABLE;
```

#### Assembler symbols for all encodings

- <c> See [C1.2.5 Standard assembler syntax fields](#) on page 309.
- <q> See [C1.2.5 Standard assembler syntax fields](#) on page 309.
- <Rt> Is the general-purpose register to be transferred, encoded in the "Rt" field.
- <Rn> Is the general-purpose base register, encoded in the "Rn" field.

#### Operation for all encodings

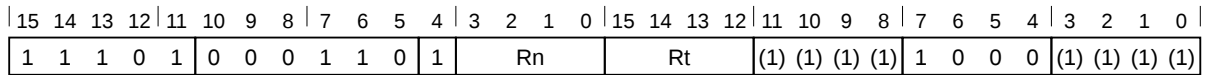
```
1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 address = R[n];
4 R[t] = MemO(address, 4);
```

### C2.4.44 LDAB

Load-Acquire Byte. Load-Acquire Byte loads a byte from memory, zero-extends it to form a 32-bit word and writes it to a register. The instruction also has memory ordering semantics.

#### T1

Armv8-M



#### T1 variant

LDAB{<c>}{<q>} <Rt>, [<Rn>]

#### Decode for this encoding

```
1 t = UInt(Rt); n = UInt(Rn);
2 if t IN {13,15} || n == 15 then UNPREDICTABLE;
```

#### Assembler symbols for all encodings

<c> See [C1.2.5 Standard assembler syntax fields](#) on page 309.  
 <q> See [C1.2.5 Standard assembler syntax fields](#) on page 309.  
 <Rt> Is the general-purpose register to be transferred, encoded in the "Rt" field.  
 <Rn> Is the general-purpose base register, encoded in the "Rn" field.

#### Operation for all encodings

```
1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 address = R[n];
4 R[t] = ZeroExtend(MemO[address, 1], 32);
```



### C2.4.45 LDAEX

Load-Acquire Exclusive Word. Load-Acquire Exclusive Word loads a word from memory, writes it to a register, and:

- If the address has the Shareable memory attribute, marks the physical address as exclusive access for the executing PE in a global monitor.
- Causes the executing PE to indicate an active exclusive access in the local monitor.

The instruction also has memory ordering semantics.

#### T1

Armv8-M

|    |    |    |    |    |    |   |   |   |   |   |   |   |   |   |   |    |    |    |    |    |    |   |   |   |   |   |   |   |   |    |   |  |  |  |  |  |  |  |  |  |  |  |  |  |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |
|----|----|----|----|----|----|---|---|---|---|---|---|---|---|---|---|----|----|----|----|----|----|---|---|---|---|---|---|---|---|----|---|--|--|--|--|--|--|--|--|--|--|--|--|--|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1  | 0 |  |  |  |  |  |  |  |  |  |  |  |  |  |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |
| 1  | 1  | 1  | 0  | 1  | 0  | 0 | 0 | 1 | 1 | 0 | 1 |   |   |   |   | Rn |    |    |    |    |    |   |   |   |   |   |   |   |   | Rt |   |  |  |  |  |  |  |  |  |  |  |  |  |  | (1) | (1) | (1) | (1) | (1) | (1) | (1) | (1) | (1) | (1) | (1) | (1) | (1) | (1) | (1) | (1) |

#### T1 variant

LDAEX{<c>}{<q>} <Rt>, [<Rn>]

#### Decode for this encoding

```
1 t = UInt(Rt); n = UInt(Rn);
2 if t IN {13,15} || n == 15 then UNPREDICTABLE;
```

#### Assembler symbols for all encodings

<c> See [C1.2.5 Standard assembler syntax fields](#) on page 309.  
 <q> See [C1.2.5 Standard assembler syntax fields](#) on page 309.  
 <Rt> Is the general-purpose register to be transferred, encoded in the "Rt" field.  
 <Rn> Is the general-purpose base register, encoded in the "Rn" field.

#### Operation for all encodings

```
1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 address = R[n];
4 SetExclusiveMonitors(address, 4);
5 R[t] = MemO[address, 4];
```

## C2.4.46 LDAEXB

Load-Acquire Exclusive Byte. Load-Acquire Exclusive Byte loads a byte from memory, zero-extends it to form a 32-bit word, writes it to a register, and:

- If the address has the Shareable memory attribute, marks the physical address as exclusive access for the executing PE in a global monitor.
- Causes the executing PE to indicate an active exclusive access in the local monitor.

The instruction also has memory ordering semantics.

### T1

Armv8-M

|    |    |    |    |    |    |   |   |   |   |   |   |   |   |   |   |    |    |    |    |    |    |   |   |   |   |   |   |   |   |    |   |  |  |  |  |  |  |  |  |  |  |  |  |  |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |
|----|----|----|----|----|----|---|---|---|---|---|---|---|---|---|---|----|----|----|----|----|----|---|---|---|---|---|---|---|---|----|---|--|--|--|--|--|--|--|--|--|--|--|--|--|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1  | 0 |  |  |  |  |  |  |  |  |  |  |  |  |  |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |
| 1  | 1  | 1  | 0  | 1  | 0  | 0 | 0 | 1 | 1 | 0 | 1 |   |   |   |   | Rn |    |    |    |    |    |   |   |   |   |   |   |   |   | Rt |   |  |  |  |  |  |  |  |  |  |  |  |  |  | (1) | (1) | (1) | (1) | (1) | (1) | (1) | (1) | (1) | (1) | (1) | (1) | (1) | (1) | (1) | (1) |

### T1 variant

LDAEXB{<c>}{<q>} <Rt>, [<Rn>]

### Decode for this encoding

```
1 t = UInt(Rt); n = UInt(Rn);
2 if t IN {13,15} || n == 15 then UNPREDICTABLE;
```

### Assembler symbols for all encodings

<c> See [C1.2.5 Standard assembler syntax fields](#) on page 309.  
 <q> See [C1.2.5 Standard assembler syntax fields](#) on page 309.  
 <Rt> Is the general-purpose register to be transferred, encoded in the "Rt" field.  
 <Rn> Is the general-purpose base register, encoded in the "Rn" field.

### Operation for all encodings

```
1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 address = R[n];
4 SetExclusiveMonitors(address, 1);
5 R[t] = ZeroExtend(MemO(address, 1), 32);
```

### C2.4.47 LDAEXH

Load-Acquire Exclusive Halfword. Load-Acquire Exclusive Halfword loads a halfword from memory, zero-extends it to form a 32-bit word, writes it to a register, and:

- If the address has the Shareable memory attribute, marks the physical address as exclusive access for the executing PE in a global monitor.
- Causes the executing PE to indicate an active exclusive access in the local monitor.

The instruction also has memory ordering semantics.

#### T1

Armv8-M

|    |    |    |    |    |    |   |   |   |   |   |   |    |   |   |   |    |    |    |    |     |     |     |     |   |   |   |   |     |     |     |     |
|----|----|----|----|----|----|---|---|---|---|---|---|----|---|---|---|----|----|----|----|-----|-----|-----|-----|---|---|---|---|-----|-----|-----|-----|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3  | 2 | 1 | 0 | 15 | 14 | 13 | 12 | 11  | 10  | 9   | 8   | 7 | 6 | 5 | 4 | 3   | 2   | 1   | 0   |
| 1  | 1  | 1  | 0  | 1  | 0  | 0 | 0 | 1 | 1 | 0 | 1 | Rn |   |   |   | Rt |    |    |    | (1) | (1) | (1) | (1) | 1 | 1 | 0 | 1 | (1) | (1) | (1) | (1) |

#### T1 variant

LDAEXH{<c>}{<q>} <Rt>, [<Rn>]

#### Decode for this encoding

```
1 t = UInt(Rt); n = UInt(Rn);
2 if t IN {13,15} || n == 15 then UNPREDICTABLE;
```

#### Assembler symbols for all encodings

- <c> See [C1.2.5 Standard assembler syntax fields](#) on page 309.  
 <q> See [C1.2.5 Standard assembler syntax fields](#) on page 309.  
 <Rt> Is the general-purpose register to be transferred, encoded in the "Rt" field.  
 <Rn> Is the general-purpose base register, encoded in the "Rn" field.

#### Operation for all encodings

```
1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 address = R[n];
4 SetExclusiveMonitors(address, 2);
5 R[t] = ZeroExtend(MemO[address, 2], 32);
```

## C2.4.48 LDAH

Load-Acquire Halfword. Load-Acquire Halfword loads a halfword from memory, zero-extends it to form a 32-bit word and writes it to a register. The instruction also has memory ordering semantics.

### T1

Armv8-M

|    |    |    |    |    |    |   |   |   |   |   |   |   |   |    |   |    |    |    |    |     |     |     |     |   |   |   |   |     |     |     |     |
|----|----|----|----|----|----|---|---|---|---|---|---|---|---|----|---|----|----|----|----|-----|-----|-----|-----|---|---|---|---|-----|-----|-----|-----|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1  | 0 | 15 | 14 | 13 | 12 | 11  | 10  | 9   | 8   | 7 | 6 | 5 | 4 | 3   | 2   | 1   | 0   |
| 1  | 1  | 1  | 0  | 1  | 0  | 0 | 0 | 1 | 1 | 0 | 1 |   |   | Rn |   |    | Rt |    |    | (1) | (1) | (1) | (1) | 1 | 0 | 0 | 1 | (1) | (1) | (1) | (1) |

### T1 variant

LDAH{<c>}{<q>} <Rt>, [<Rn>]

### Decode for this encoding

```
1 t = UInt(Rt); n = UInt(Rn);
2 if t IN {13,15} || n == 15 then UNPREDICTABLE;
```

### Assembler symbols for all encodings

<c> See [C1.2.5 Standard assembler syntax fields](#) on page 309.  
 <q> See [C1.2.5 Standard assembler syntax fields](#) on page 309.  
 <Rt> Is the general-purpose register to be transferred, encoded in the "Rt" field.  
 <Rn> Is the general-purpose base register, encoded in the "Rn" field.

### Operation for all encodings

```
1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 address = R[n];
4 R[t] = ZeroExtend(MemO[address, 2], 32);
```

### C2.4.49 LDC, LDC2 (immediate)

Load Coprocessor (immediate). Load Coprocessor loads memory data from a sequence of consecutive memory addresses to a coprocessor. If no coprocessor can execute the instruction, a UsageFault exception is generated.

This is a generic coprocessor instruction. Some of the fields have no functionality defined by the architecture and are free for use by the coprocessor instruction set designer. These fields are the D bit, the CRd field, and in the Unindexed addressing mode only, the imm8 field.

Arm reserves coprocessors CP8 to CP15, and this manual defines the valid instructions when coproc is in this range.

#### T1

*Armv8-M Main Extension only*

|    |    |    |    |    |    |   |   |   |   |   |   |            |   |   |   |     |    |    |    |        |    |   |   |      |   |   |   |   |   |   |   |
|----|----|----|----|----|----|---|---|---|---|---|---|------------|---|---|---|-----|----|----|----|--------|----|---|---|------|---|---|---|---|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3          | 2 | 1 | 0 | 15  | 14 | 13 | 12 | 11     | 10 | 9 | 8 | 7    | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 1  | 1  | 1  | 0  | 1  | 1  | 0 | P | U | D | W | 1 | Rn != 1111 |   |   |   | CRd |    |    |    | coproc |    |   |   | imm8 |   |   |   |   |   |   |   |

#### Offset variant

Applies when **P == 1 && W == 0**.

LDC{L}{<c>}{<q>} <coproc>, <CRd>, [<Rn>{, #+/-}<imm>]

#### Post-indexed variant

Applies when **P == 0 && W == 1**.

LDC{L}{<c>}{<q>} <coproc>, <CRd>, [<Rn>], #+/-<imm>

#### Pre-indexed variant

Applies when **P == 1 && W == 1**.

LDC{L}{<c>}{<q>} <coproc>, <CRd>, [<Rn>, #+/-<imm>]!

#### Unindexed variant

Applies when **P == 0 && U == 1 && W == 0**.

LDC{L}{<c>}{<q>} <coproc>, <CRd>, [<Rn>], <option>

#### Decode for this encoding

```

1 if Rn == '1111' then SEE "LDC (literal)";
2 if P == '0' && U == '0' && D == '1' && W == '0' then SEE "MRRC, MRRC2";
3 if coproc IN '101x' then SEE "Floating-point";
4 if P == '0' && U == '0' && D == '0' && W == '0' then UNDEFINED;
5 if !HaveMainExt() then UNDEFINED;
6 n = UInt(Rn); cp = UInt(coproc); imm32 = ZeroExtend(imm8:'00', 32);
7 index = (P == '1'); add = (U == '1'); wback = (W == '1');
```

#### T2

*Armv8-M Main Extension only*

|    |    |    |    |    |    |   |   |   |   |   |   |            |   |   |   |     |    |    |    |        |    |   |   |      |   |   |   |   |   |   |   |
|----|----|----|----|----|----|---|---|---|---|---|---|------------|---|---|---|-----|----|----|----|--------|----|---|---|------|---|---|---|---|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3          | 2 | 1 | 0 | 15  | 14 | 13 | 12 | 11     | 10 | 9 | 8 | 7    | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 1  | 1  | 1  | 1  | 1  | 1  | 0 | P | U | D | W | 1 | Rn != 1111 |   |   |   | CRd |    |    |    | coproc |    |   |   | imm8 |   |   |   |   |   |   |   |

#### Offset variant

**Applies when P == 1 && W == 0.**

LDC2{L}{<c>}{<q>} <coproc>, <CRd>, [<Rn>{, #+/-}<imm>}]

**Post-indexed variant**

**Applies when P == 0 && W == 1.**

LDC2{L}{<c>}{<q>} <coproc>, <CRd>, [<Rn>], #+/-<imm>

**Pre-indexed variant**

**Applies when P == 1 && W == 1.**

LDC2{L}{<c>}{<q>} <coproc>, <CRd>, [<Rn>, #+/-<imm>!]

**Unindexed variant**

**Applies when P == 0 && U == 1 && W == 0.**

LDC2{L}{<c>}{<q>} <coproc>, <CRd>, [<Rn>], <option>

### Decode for this encoding

```

1 if Rn == '1111' then SEE "LDC (literal)";
2 if P == '0' && U == '0' && D == '1' && W == '0' then SEE "MRRC, MRRC2";
3 if coproc IN '101x' then SEE "Floating-point";
4 if P == '0' && U == '0' && D == '0' && W == '0' then UNDEFINED;
5 if !HaveMainExt() then UNDEFINED;
6 n = UInt(Rn); cp = UInt(coproc); imm32 = ZeroExtend(imm8:'00', 32);
7 index = (P == '1'); add = (U == '1'); wback = (W == '1');
```

### Assembler symbols for all encodings

|          |                                                                                                                                                                                                                                                        |
|----------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| L        | If specified, selects the D == 1 form of the encoding. If omitted, selects the D == 0 form.                                                                                                                                                            |
| <c>      | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                                                                                                                                                                               |
| <q>      | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                                                                                                                                                                               |
| <coproc> | Is the name of the coprocessor, encoded in the "coproc" field. The generic coprocessor names are p0 to p15.                                                                                                                                            |
| <CRd>    | Is the coprocessor register to be transferred, encoded in the "CRd" field.                                                                                                                                                                             |
| <Rn>     | Is the general-purpose base register, encoded in the "Rn" field. If the PC is used, see <a href="#">C2.4.50 LDC, LDC2 (literal)</a> on page 444.                                                                                                       |
| <option> | Is a coprocessor option, in the range 0 to 255 enclosed in { }, encoded in the "imm8" field.                                                                                                                                                           |
| +/-      | Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in the "U" field. It can have the following values: <ul style="list-style-type: none"> <li>- when U = 0</li> <li>+ when U = 1</li> </ul> |
| <imm>    | Is the immediate offset used for forming the address, a multiple of 4 in the range 0-1020, defaulting to 0 and encoded in the "imm8" field, as <imm>/4.                                                                                                |

### Operation for all encodings

```

1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 ExecuteCPCheck(cp);
4 if !Coprocc_Accepted(cp, ThisInstr()) then
5 GenerateCoproccorException();
6 else
7 offset_addr = if add then (R[n] + imm32) else (R[n] - imm32);
8 address = if index then offset_addr else R[n];
9
```

```
10 // Determine if the stack pointer limit check should be performed
11 if wback && n == 13 then
12 (limit, applylimit) = LookUpSPLim(LookUpSP());
13 else
14 applylimit = FALSE;
15
16 // Memory operation only performed if limit not violated
17 if !applylimit || (UInt(offset_addr) >= UInt(limit)) then
18 repeat
19 Coproc_SendLoadedWord(MemA[address,4], cp, ThisInstr());
20 address = address + 4;
21 until Coproc_DoneLoading(cp, ThisInstr());
22
23 // If the stack pointer is being updated a fault will be raised
24 // if the limit is violated
25 if wback then RSPCheck[n] = offset_addr;
```

## C2.4.50 LDC, LDC2 (literal)

Load Coprocessor (literal). Load Coprocessor loads memory data from a sequence of consecutive memory addresses to a coprocessor. If no coprocessor can execute the instruction, a UsageFault exception is generated.

This is a generic coprocessor instruction. The D bit and the CRd field have no functionality defined by the architecture and are free for use by the coprocessor instruction set designer.

Arm reserves coprocessors CP8 to CP15, and this manual defines the valid instructions when coproc is in this range.

### T1

*Armv8-M Main Extension only*

|    |    |    |    |    |    |   |   |   |   |   |   |   |   |   |   |     |        |    |    |    |    |   |      |   |   |   |   |   |   |   |   |
|----|----|----|----|----|----|---|---|---|---|---|---|---|---|---|---|-----|--------|----|----|----|----|---|------|---|---|---|---|---|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | 15  | 14     | 13 | 12 | 11 | 10 | 9 | 8    | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 1  | 1  | 1  | 0  | 1  | 1  | 0 | P | U | D | W | 1 | 1 | 1 | 1 | 1 | CRd | coproc |    |    |    |    |   | imm8 |   |   |   |   |   |   |   |   |

#### T1 variant

**Applies when  $!(P == 0 \ \&\& \ U == 0 \ \&\& \ W == 0)$ .**

LDC{L}{<c>}{<q>} <coproc>, <CRd>, <label>  
 LDC{L}{<c>}{<q>} <coproc>, <CRd>, [PC, #{+/-}<imm>]

#### Decode for this encoding

```

1 if P == '0' && U == '0' && D == '1' && W == '0' then SEE "MRRC, MRRC2";
2 if coproc IN '101x' then SEE "Floating-point";
3 if P == '0' && U == '0' && D == '0' && W == '0' then UNDEFINED;
4 if !HaveMainExt() then UNDEFINED;
5 index = (P == '1'); // Always TRUE in the T32 instruction set
6 add = (U == '1'); cp = UInt(coproc); imm32 = ZeroExtend(imm8:'00', 32);
7 if W == '1' || P == '0' then UNPREDICTABLE;

```

#### CONSTRAINED UNPREDICTABLE behavior

If  $W == '1' \ || \ P == '0'$ , then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes without writeback of the base address.
- The instruction executes as LDC with writeback to the PC.

### T2

*Armv8-M Main Extension only*

|    |    |    |    |    |    |   |   |   |   |   |   |   |   |   |   |     |        |    |    |    |    |   |      |   |   |   |   |   |   |   |   |
|----|----|----|----|----|----|---|---|---|---|---|---|---|---|---|---|-----|--------|----|----|----|----|---|------|---|---|---|---|---|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | 15  | 14     | 13 | 12 | 11 | 10 | 9 | 8    | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 1  | 1  | 1  | 1  | 1  | 1  | 0 | P | U | D | W | 1 | 1 | 1 | 1 | 1 | CRd | coproc |    |    |    |    |   | imm8 |   |   |   |   |   |   |   |   |

#### T2 variant

**Applies when  $!(P == 0 \ \&\& \ U == 0 \ \&\& \ W == 0)$ .**

LDC2{L}{<c>}{<q>} <coproc>, <CRd>, <label>  
 LDC2{L}{<c>}{<q>} <coproc>, <CRd>, [PC, #{+/-}<imm>]

#### Decode for this encoding



```

1 if P == '0' && U == '0' && D == '1' && W == '0' then SEE "MRRC, MRRC2";
2 if P == '0' && U == '0' && D == '0' && W == '0' then UNDEFINED;
3 if coproc IN '101x' then UNDEFINED;
4 if !HaveMainExt() then UNDEFINED;
5 index = (P == '1'); // Always TRUE in the T32 instruction set
6 add = (U == '1'); cp = UInt(coproc); imm32 = ZeroExtend(imm8:'00', 32);
7 if W == '1' || P == '0' then UNPREDICTABLE;

```

### CONSTRAINED UNPREDICTABLE behavior

If `W == '1' || P == '0'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes without writeback of the base address.
- The instruction executes as LDC with writeback to the PC.

### Assembler symbols for all encodings

|          |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                    |
|----------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| L        | If specified, selects the <code>D == 1</code> form of the encoding. If omitted, selects the <code>D == 0</code> form.                                                                                                                                                                                                                                                                                                                                                                                                                                                              |
| <c>      | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |
| <q>      | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |
| <coproc> | Is the name of the coprocessor, encoded in the "coproc" field. The generic coprocessor names are p0 to p15.                                                                                                                                                                                                                                                                                                                                                                                                                                                                        |
| <CRd>    | Is the coprocessor register to be transferred, encoded in the "CRd" field.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |
| <label>  | The label of the literal data item that is to be loaded into <Rt>. The assembler calculates the required value of the offset from the <a href="#">Align</a> (PC, 4) value of the instruction to this label. Permitted values of the offset are multiples of 4 in the range -1020 to 1020. If the offset is zero or positive, <code>imm32</code> is equal to the offset and <code>add == TRUE</code> (encoded as <code>U == 1</code> ). If the offset is negative, <code>imm32</code> is equal to minus the offset and <code>add == FALSE</code> (encoded as <code>U == 0</code> ). |
| +/-      | Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in the "U" field. It can have the following values: <ul style="list-style-type: none"> <li>- when <code>U = 0</code></li> <li>+ when <code>U = 1</code></li> </ul>                                                                                                                                                                                                                                                                                                   |
| <imm>    | Is the immediate offset used for forming the address, a multiple of 4 in the range 0-1020, defaulting to 0 and encoded in the "imm8" field, as <code>&lt;imm&gt;/4</code> .                                                                                                                                                                                                                                                                                                                                                                                                        |

### Operation for all encodings

```

1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 ExecuteCPCheck(cp);
4 if !CoproccAccepted(cp, ThisInstr()) then
5 GenerateCoproccorException();
6 else
7 offset_addr = if add then (Align(PC,4) + imm32) else (Align(PC,4) - imm32);
8 address = if index then offset_addr else Align(PC,4);
9 repeat
10 Coproc_SendLoadedWord(MemA[address,4], cp, ThisInstr()); address = address + 4;
11 until Coproc_DoneLoading(cp, ThisInstr());

```

## C2.4.51 LDM, LDMIA, LDMFD

Load Multiple (Increment After, Full Descending). Load Multiple loads multiple registers from consecutive memory locations using an address from a base register. The sequential memory locations start at this address, and the address just above the last of those locations can optionally be written back to the base register.

The registers loaded can include the PC. If they do, the word loaded for the PC is treated as a branch address, a function return value, or an exception return value. Bit[0] of the address in the PC complies with the Arm architecture interworking rules for switching between the A32 and T32 instruction sets. However, Armv8-M only supports the T32 instruction set, so bit[0] must be 1. If bit[0] of the target address is 0, and the target address is not FNC\_RETURN or EXC\_RETURN, the PE takes an INVSTATE UsageFault exception on the instruction at the target address.

This instruction is used by the alias [POP \(multiple registers\)](#).

### T1

Armv8-M

|    |    |    |    |    |    |   |   |   |               |   |   |   |   |   |   |  |
|----|----|----|----|----|----|---|---|---|---------------|---|---|---|---|---|---|--|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6             | 5 | 4 | 3 | 2 | 1 | 0 |  |
| 1  | 1  | 0  | 0  | 1  | Rn |   |   |   | register_list |   |   |   |   |   |   |  |

#### T1 variant

```
LDM{IA}{<c>}{<q>} <Rn>{!}, <registers>
// Preferred syntax
LDMFD{<c>}{<q>} <Rn>{!}, <registers>
// Alternate syntax, Full Descending stack
```

#### Decode for this encoding

```
1 n = UInt(Rn); registers = '00000000':register_list; wback = (registers[n] == '0');
2 if BitCount(registers) < 1 then UNPREDICTABLE;
```

#### CONSTRAINED UNPREDICTABLE behavior

If `BitCount(registers) < 1`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes as LDM with the same addressing mode but targeting an unspecified set of registers. These registers might include R15. If the instruction specifies writeback, the modification to the base address on writeback might differ from the number of registers loaded.

### T2

Armv8-M Main Extension only

|    |    |    |    |    |    |   |   |   |   |   |   |    |   |   |   |    |    |     |               |    |    |   |   |   |   |   |   |   |   |   |   |
|----|----|----|----|----|----|---|---|---|---|---|---|----|---|---|---|----|----|-----|---------------|----|----|---|---|---|---|---|---|---|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3  | 2 | 1 | 0 | 15 | 14 | 13  | 12            | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 1  | 1  | 1  | 0  | 1  | 0  | 0 | 0 | 1 | 0 | W | 1 | Rn |   |   |   | P  | M  | (0) | register_list |    |    |   |   |   |   |   |   |   |   |   |   |

#### T2 variant

```
LDM{IA}{<c>}.W <Rn>{!}, <registers>
// Preferred syntax
// if <Rn>, '!' and <registers> can be represented in T1
LDMFD{<c>}.W <Rn>{!}, <registers>
```

```
// Alternate syntax
// Full Descending stack, if <Rn>, '!' and <registers> can be represented in T1
LDM{IA}{<c>}{<q>} <Rn>{!}, <registers>
// Preferred syntax
LDMFD{<c>}{<q>} <Rn>{!}, <registers>
// Alternate syntax, Full Descending stack
```

### Decode for this encoding

```
1 if !HaveMainExt() then UNDEFINED;
2 n = UInt(Rn); registers = P:M:'0':register_list; wback = (W == '1');
3 if n == 15 || BitCount(registers) < 2 || (P == '1' && M == '1') then UNPREDICTABLE;
4 if registers[15] == '1' && InITBlock() && !LastInITBlock() then UNPREDICTABLE;
5 if wback && registers[n] == '1' then UNPREDICTABLE;
```

### CONSTRAINED UNPREDICTABLE behavior

If `BitCount(registers) < 1`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes as LDM with the same addressing mode but targeting an unspecified set of registers. These registers might include R15. If the instruction specifies writeback, the modification to the base address on writeback might differ from the number of registers loaded.

### CONSTRAINED UNPREDICTABLE behavior

If `wback && registers<n> == '1'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such an instruction, the base address might be corrupted so that the instruction cannot be repeated.

### CONSTRAINED UNPREDICTABLE behavior

If `BitCount(registers) == 1`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction loads a single register using the specified addressing modes.
- The instruction executes as LDM with the same addressing mode but targeting an unspecified set of registers. These registers might include R15.

### CONSTRAINED UNPREDICTABLE behavior

If `P == '1' && M == '1'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction loads the register list and either R14 or R15, both R14 and R15, or neither of these registers.

## T3

Armv8-M

|    |    |    |    |    |    |   |   |               |   |   |   |   |   |   |   |
|----|----|----|----|----|----|---|---|---------------|---|---|---|---|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7             | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 1  | 0  | 1  | 1  | 1  | 1  | 0 | P | register_list |   |   |   |   |   |   |   |

### T3 variant

LDM{<c>}{<q>} SP!, <registers>

### Decode for this encoding

```

1 n = 13; wback = TRUE;
2 registers = P:'0000000':register_list;
3 if BitCount(registers) < 1 then UNPREDICTABLE;
4 if registers[15] == '1' && InITBlock() && !LastInITBlock() then UNPREDICTABLE;
```

### CONSTRAINED UNPREDICTABLE behavior

If `BitCount(registers) < 1`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes as LDM with the same addressing mode but targeting an unspecified set of registers. These registers might include R15. If the instruction specifies writeback, the modification to the base address on writeback might differ from the number of registers loaded.

### Assembler symbols for all encodings

|             |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                        |
|-------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| IA          | Is an optional suffix for the Increment After form.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                    |
| <c>         | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                               |
| <q>         | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                               |
| <Rn>        | Is the general-purpose base register, encoded in the "Rn" field.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                       |
| !           | For encoding T1: the address adjusted by the size of the data loaded is written back to the base register. It is omitted if <Rn> is included in <registers>, otherwise it must be present.<br>For encoding T2: the address adjusted by the size of the data loaded is written back to the base register. If specified, it is encoded in the "W" field as 1, otherwise this field defaults to 0.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                        |
| <registers> | For encoding T1: is a list of one or more registers to be loaded, separated by commas and surrounded by { and }. The registers in the list must be in the range R0-R7, encoded in the "register_list" field. For encoding T2: is a list of one or more registers to be loaded, separated by commas and surrounded by { and }. The registers in the list must be in the range R0-R12, encoded in the "register_list" field, and can optionally contain one of the LR or the PC. If the LR is in the list, the "M" field is set to 1, otherwise it defaults to 0. If the PC is in the list, the "P" field is set to 1, otherwise it defaults to 0. If the PC is in the list: <ul style="list-style-type: none"> <li>- The LR must not be in the list.</li> <li>- The instruction must be either outside any IT block, or the last instruction in an IT block.</li> </ul> For encoding T3: is a list of one or more registers to be loaded, separated by commas and surrounded by { and }. The registers in the list must be in the range R0-R7, encoded in the "register_list" field, and can optionally include the PC. If the PC is in the list, the "P" field is set to 1, otherwise this field defaults to 0. If the PC is in the list, the instruction must be either outside any IT block, or the last instruction in an IT block. |

### Operation for all encodings

```

1 if ConditionPassed() then
2 EncodingSpecificOperations();
```

```

3 address = R[n];
4 if n == 13 && wback then
5 (limit, applylimit) = LookUpSPLim(LookUpSP());
6 // If memory operation is not performed as a result of a stack limit violation,
7 // and the write-back of the SP itself does not raise a stack limit violation, it
8 // is "IMPLEMENTATION_DEFINED" whether a SPLIM exception is raised.
9 // Arm recommends that any instruction which discards a memory access as
10 // a result of a stack limit violation, and where the write-back of the SP itself
11 // does not raise a stack limit violation, generates an SPLIM exception.
12 if boolean IMPLEMENTATION_DEFINED "SPLIM exception on invalid memory access" then
13 if applylimit && (UInt(address) < UInt(limit)) then
14 if HaveMainExt() then
15 UFSR.STKOF = '1';
16 // If Main Extension is not implemented the fault always escalates to
17 // HardFault
18 excInfo = CreateException(UsageFault, FALSE, boolean UNKNOWN);
19 HandleException(excInfo);
20 else
21 applylimit = FALSE;
22
23 for i = 0 to 14
24 // If R[n] is the SP, memory operation only performed if limit not violated
25 if registers[i] == '1' && (!applylimit || (UInt(address) >= UInt(limit))) then
26 if i != n then
27 R[i] = MemA[address,4];
28 else
29 newBaseVal = MemA[address,4];
30 address = address + 4;
31 if registers[15] == '1' && (!applylimit || (UInt(address) >= UInt(limit))) then
32 newPCVal = MemA[address,4];
33
34 // If the register list contains the register that holds the base address it
35 // must be updated after all memory reads have been performed. This prevents
36 // the base address being overwritten if one of the memory reads generates a
37 // fault.
38 if registers[n] == '1' then
39 wback = TRUE;
40 else
41 newBaseVal = R[n] + 4*BitCount(registers);
42 // If the PC is in the register list update that now, which might raise a fault
43 // Likewise if R[n] is the SP writing back might raise a fault due to SP limit violation
44 if registers[15] == '1' then
45 LoadWritePC(newPCVal, n, newBaseVal, wback, FALSE);
46 elseif wback then
47 RSPCheck[n] = newBaseVal;

```

## C2.4.52 LDMDB, LDMEA

Load Multiple Decrement Before (Empty Ascending). Load Multiple Decrement Before (Load Multiple Empty Ascending) loads multiple registers from sequential memory locations using an address from a base register. The sequential memory locations end just below this address, and the address of the first of those locations can optionally be written back to the base register.

The registers loaded can include the PC. If they do, the word loaded for the PC is treated as a branch address or an exception return value. Bit[0] complies with the Arm architecture interworking rules for switching between the A32 and T32 instruction sets. However, Armv8-M only supports the T32 instruction set, so bit[0] must be 1. If bit[0] is 0 the PE takes an INVSTATE UsageFault exception on the instruction at the target address.

### T1

Armv8-M Main Extension only

|    |    |    |    |    |    |   |   |   |   |   |   |    |   |   |   |    |    |     |               |    |    |   |   |   |   |   |   |   |   |   |   |
|----|----|----|----|----|----|---|---|---|---|---|---|----|---|---|---|----|----|-----|---------------|----|----|---|---|---|---|---|---|---|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3  | 2 | 1 | 0 | 15 | 14 | 13  | 12            | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 1  | 1  | 1  | 0  | 1  | 0  | 0 | 1 | 0 | 0 | W | 1 | Rn |   |   |   | P  | M  | (0) | register_list |    |    |   |   |   |   |   |   |   |   |   |   |

### T1 variant

```
LDMDB{<c>}{<q>} <Rn>{!}, <registers>
// Preferred syntax
LDMEA{<c>}{<q>} <Rn>{!}, <registers>
// Alternate syntax, Empty Ascending stack
```

### Decode for this encoding

```
1 if !HaveMainExt() then UNDEFINED;
2 n = UInt(Rn); registers = P:M:'0':register_list; wback = (W == '1');
3 if n == 15 || BitCount(registers) < 2 || (P == '1' && M == '1') then UNPREDICTABLE;
4 if registers[15] == '1' && InITBlock() && !LastInITBlock() then UNPREDICTABLE;
5 if wback && registers[n] == '1' then UNPREDICTABLE;
```

### CONSTRAINED UNPREDICTABLE behavior

If `wback && registers<n> == '1'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such an instruction, the base address might be corrupted so that the instruction cannot be repeated.

### CONSTRAINED UNPREDICTABLE behavior

If `BitCount(registers) < 1`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes as LDM with the same addressing mode but targeting an unspecified set of registers. These registers might include R15. If the instruction specifies writeback, the modification to the base address on writeback might differ from the number of registers loaded.

### CONSTRAINED UNPREDICTABLE behavior

If `BitCount(registers) == 1`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction loads a single register using the specified addressing modes.
- The instruction executes as LDM with the same addressing mode but targeting an unspecified set of registers. These registers might include R15.

### CONSTRAINED UNPREDICTABLE behavior

If `P == '1' && M == '1'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction loads the register list and either R14 or R15, both R14 and R15, or neither of these registers.

### Assembler symbols for all encodings

|                                |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                          |
|--------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>&lt;c&gt;</code>         | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                 |
| <code>&lt;q&gt;</code>         | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                 |
| <code>&lt;Rn&gt;</code>        | Is the general-purpose base register, encoded in the "Rn" field.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |
| <code>!</code>                 | The address adjusted by the size of the data loaded is written back to the base register. If specified, it is encoded in the "W" field as 1, otherwise this field defaults to 0.                                                                                                                                                                                                                                                                                                                                                                                                                                                         |
| <code>&lt;registers&gt;</code> | Is a list of one or more registers to be loaded, separated by commas and surrounded by { and }. The registers in the list must be in the range R0-R12, encoded in the "register_list" field, and can optionally contain one of the LR or the PC. If the LR is in the list, the "M" field is set to 1, otherwise it defaults to 0. If the PC is in the list, the "P" field is set to 1, otherwise it defaults to 0. If the PC is in the list: <ul style="list-style-type: none"> <li>- The LR must not be in the list.</li> <li>- The instruction must be either outside any IT block, or the last instruction in an IT block.</li> </ul> |

### Operation for all encodings

```

1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 address = R[n] - 4*BitCount(registers);
4
5 // Determine if the stack pointer limit should be checked
6 if n == 13 && wback && registers[n] == '0' then
7 (limit, applylimit) = LookUpSPLim(LookUpSP());
8 doOperation = (!applylimit || (UInt(address) >= UInt(limit)));
9 else
10 doOperation = TRUE;
11
12 for i = 0 to 15
13 // Memory operation only performed if limit not violated
14 if registers[i] == '1' && doOperation then
15 data = MemA[address,4];
16 address = address + 4;
17 if i == 15 then
18 newPCVal = data;
19 elseif i == n then
20 newBaseVal = data;
21 else
22 R[i] = data;
23
24 // If the register list contains the register that holds the base address it
25 // must be updated after all memory reads have been performed. This prevents
26 // the base address being overwritten if one of the memory reads generates a
27 // fault.
28 if registers[n] == '1' then

```

```
29 wback = TRUE;
30 else
31 newBaseVal = R[n] - 4*BitCount(registers);
32 // If the PC is in the register list update that now, which may raise a fault
33 if registers[15] == '1' then
34 LoadWritePC(newPCVal, n, newBaseVal, wback, TRUE);
35 elsif wback then
36 RSPCheck[n] = newBaseVal;
```



### C2.4.53 LDR (immediate)

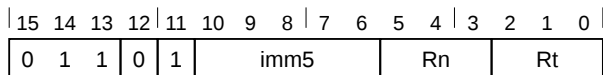
Load Register (immediate). Load Register (immediate) calculates an address from a base register value and an immediate offset, loads a word from memory, and writes it to a register. It can use offset, post-indexed, or pre-indexed addressing.

The register loaded can be the PC. If it is, the word loaded for the PC is treated as a branch address or an exception return value. Bit[0] complies with the Arm architecture interworking rules for switching between the A32 and T32 instruction sets. However, Armv8-M only supports the T32 instruction set, so bit[0] must be 1. If bit[0] is 0 the PE takes an INVSTATE UsageFault exception on the instruction at the target address.

This instruction is used by the alias [POP \(single register\)](#).

#### T1

Armv8-M



#### T1 variant

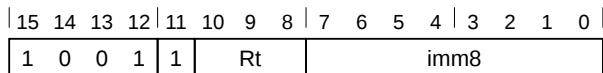
LDR{<c>}{<q>} <Rt>, [<Rn> {, #<+><imm>}]

#### Decode for this encoding

```
1 t = UInt(Rt); n = UInt(Rn); imm32 = ZeroExtend(imm5:'00', 32);
2 index = TRUE; add = TRUE; wback = FALSE;
```

#### T2

Armv8-M



#### T2 variant

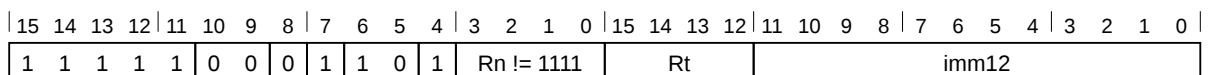
LDR{<c>}{<q>} <Rt>, [SP{, #<+><imm>}]

#### Decode for this encoding

```
1 t = UInt(Rt); n = 13; imm32 = ZeroExtend(imm8:'00', 32);
2 index = TRUE; add = TRUE; wback = FALSE;
```

#### T3

Armv8-M Main Extension only



#### T3 variant

LDR{<c>}.W <Rt>, [<Rn> {, #<+><imm>}]  
 // <Rt>, <Rn>, <imm> can be represented in T1 or T2  
 LDR{<c>}{<q>} <Rt>, [<Rn> {, #<+><imm>}]

#### Decode for this encoding

```

1 if Rn == '1111' then SEE "LDR (literal)";
2 if !HaveMainExt() then UNDEFINED;
3 t = UInt(Rt); n = UInt(Rn); imm32 = ZeroExtend(imm12, 32); index = TRUE; add = TRUE;
4 wback = FALSE; if t == 15 && InITBlock() && !LastInITBlock() then UNPREDICTABLE;

```

## T4

Armv8-M Main Extension only

|    |    |    |    |    |    |   |   |   |   |   |   |            |   |   |   |    |    |    |    |    |    |   |   |      |   |   |   |   |   |   |   |
|----|----|----|----|----|----|---|---|---|---|---|---|------------|---|---|---|----|----|----|----|----|----|---|---|------|---|---|---|---|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3          | 2 | 1 | 0 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7    | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 1  | 1  | 1  | 1  | 1  | 0  | 0 | 0 | 0 | 1 | 0 | 1 | Rn != 1111 |   |   |   | Rt |    |    |    | 1  | P  | U | W | imm8 |   |   |   |   |   |   |   |

### Offset variant

Applies when **P == 1 && U == 0 && W == 0**.

LDR{<c>}{<q>} <Rt>, [<Rn> {, #-<imm>}]

### Post-indexed variant

Applies when **P == 0 && W == 1**.

LDR{<c>}{<q>} <Rt>, [<Rn>], #{+/-}<imm>

### Pre-indexed variant

Applies when **P == 1 && W == 1**.

LDR{<c>}{<q>} <Rt>, [<Rn>, #{+/-}<imm>]!

### Decode for this encoding

```

1 if Rn == '1111' then SEE "LDR (literal)";
2 if P == '1' && U == '1' && W == '0' then SEE LDRT;
3 if P == '0' && W == '0' then UNDEFINED;
4 if !HaveMainExt() then UNDEFINED;
5 t = UInt(Rt); n = UInt(Rn);
6 imm32 = ZeroExtend(imm8, 32); index = (P == '1'); add = (U == '1'); wback = (W == '1');
7 if (wback && n == t) || (t == 15 && InITBlock() && !LastInITBlock()) then UNPREDICTABLE;

```

### CONSTRAINED UNPREDICTABLE behavior

If **wback && n == t**, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such an instruction, the base address might be corrupted so that the instruction cannot be repeated.

### Alias conditions

| Alias                                 | preferred when                                 |
|---------------------------------------|------------------------------------------------|
| <a href="#">POP (single register)</a> | Rn == '1101' && U == '1' && imm8 == '00000100' |

### Assembler symbols for all encodings

<c> See [C1.2.5 Standard assembler syntax fields](#) on page 309.  
<q> See [C1.2.5 Standard assembler syntax fields](#) on page 309.

|                          |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                  |
|--------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>&lt;Rt&gt;</code>  | <p>For encoding T1 and T2: is the general-purpose register to be transferred, encoded in the "Rt" field.</p> <p>For encoding T3: is the general-purpose register to be transferred, encoded in the "Rt" field. The SP can be used. The PC can be used, provided the instruction is either outside an IT block or the last instruction of an IT block. If the PC is used, the instruction branches to the address (data) loaded to the PC.</p> <p>For encoding T4: is the general-purpose register to be transferred, encoded in the "Rt" field. The PC can be used, provided the instruction is either outside an IT block or the last instruction of an IT block. If the PC is used, the instruction branches to the address (data) loaded to the PC.</p>                                                                                                                       |
| <code>&lt;Rn&gt;</code>  | <p>For encoding T1: is the general-purpose base register, encoded in the "Rn" field.</p> <p>For encoding T3 and T4: is the general-purpose base register, encoded in the "Rn" field. For PC use see <a href="#">C2.4.54 LDR (literal) on page 457</a>.</p>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                       |
| <code>+/-</code>         | <p>Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in the "U" field. It can have the following values:</p> <ul style="list-style-type: none"> <li>- when U = 0</li> <li>+ when U = 1</li> </ul>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                    |
| <code>+</code>           | <p>Specifies the offset is added to the base register.</p>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                       |
| <code>&lt;imm&gt;</code> | <p>For the post-indexed or pre-indexed variant: is an 8-bit unsigned immediate byte offset, in the range 0 to 255, encoded in the "imm8" field.</p> <p>For encoding T1: is the optional positive unsigned immediate byte offset, a multiple of 4 in the range 0 to 124, defaulting to 0 and encoded in the "imm5" field as <code>&lt;imm&gt;/4</code>.</p> <p>For encoding T2: is the optional positive unsigned immediate byte offset, a multiple of 4, in the range 0 to 1020, defaulting to 0 and encoded in the "imm8" field as <code>&lt;imm&gt;/4</code>.</p> <p>For encoding T3: is an optional 12-bit unsigned immediate byte offset, in the range 0 to 4095, defaulting to 0 and encoded in the "imm12" field.</p> <p>For encoding T4: is an optional 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 and encoded in the "imm8" field.</p> |

### Operation for all encodings

```

1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 offset_addr = if add then (R[n] + imm32) else (R[n] - imm32);
4 address = if index then offset_addr else R[n];
5
6 // Determine if the stack pointer limit should be checked
7 if n == 13 && wback then
8 (limit, applylimit) = LookUpSPLim(LookUpSP());
9 else
10 applylimit = FALSE;
11 // Memory operation only performed if limit not violated
12 if !applylimit || (UInt(offset_addr) >= UInt(limit)) then
13 data = MemU[address,4];
14
15 // If the stack pointer is being updated a fault will be raised if
16 // the limit is violated
17 if t == 15 then
18 if address[1:0] == '00' then
19 LoadWritePC(data, n, offset_addr, wback, TRUE);
20 else
21 UNPREDICTABLE;
22 else
23 if wback then RSPCheck[n] = offset_addr;
24 R[t] = data;

```

### CONSTRAINED UNPREDICTABLE behavior

If `t == 15 && address<1:0> != '00'`, then one of the following behaviors must occur:

- The instruction executes as described, with no change to its behavior and no additional side effects.

- The instruction generates an UNALIGNED UsageFault.

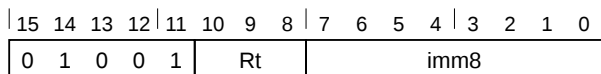
### C2.4.54 LDR (literal)

Load Register (literal). Load Register (literal) calculates an address from the PC value and an immediate offset, loads a word from memory, and writes it to a register.

The register loaded can be the PC. If it is, the word loaded for the PC is treated as a branch address or an exception return value. Bit[0] complies with the Arm architecture interworking rules for switching between the A32 and T32 instruction sets. However, Armv8-M only supports the T32 instruction set, so bit[0] must be 1. If bit[0] is 0 the PE takes an INVSTATE UsageFault exception on the instruction at the target address.

#### T1

Armv8-M



#### T1 variant

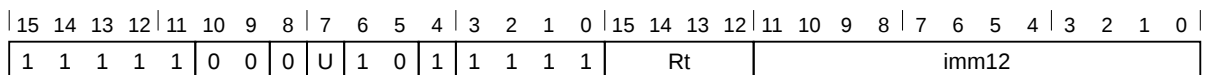
```
LDR{<c>}{<q>} <Rt>, <label>
// Normal form
```

#### Decode for this encoding

```
1 t = UInt(Rt); imm32 = ZeroExtend(imm8:'00', 32); add = TRUE;
```

#### T2

Armv8-M Main Extension only



#### T2 variant

```
LDR{<c>}.W <Rt>, <label>
// Preferred syntax, and <Rt>, <label> can be represented in T1
LDR{<c>}{<q>} <Rt>, <label>
// Preferred syntax
LDR{<c>}{<q>} <Rt>, [PC, #{+/-}<imm>]
// Alternative syntax
```

#### Decode for this encoding

```
1 if !HaveMainExt() then UNDEFINED;
2 t = UInt(Rt); imm32 = ZeroExtend(imm12, 32); add = (U == '1');
3 if t == 15 && InITBlock() && !LastInITBlock() then UNPREDICTABLE;
```

#### Assembler symbols for all encodings

<c> See [C1.2.5 Standard assembler syntax fields](#) on page 309.  
 <q> See [C1.2.5 Standard assembler syntax fields](#) on page 309.  
 <Rt> For encoding T1: is the general-purpose register to be transferred, encoded in the "Rt" field.  
 For encoding T2: is the general-purpose register to be transferred, encoded in the "Rt" field.  
 The SP can be used. The PC can be used, provided the instruction is either outside an IT block or the last instruction of an IT block. If the PC is used, the instruction branches to the address (data) loaded to the PC.

|         |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                              |
|---------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <label> | For encoding T1: the label of the literal data item that is to be loaded into <Rt>. The assembler calculates the required value of the offset from the <code>Align(PC, 4)</code> value of the instruction to this label. Permitted values of the offset are Multiples of four in the range 0 to 1020.<br>For encoding T2: the label of the literal data item that is to be loaded into <Rt>. The assembler calculates the required value of the offset from the <code>Align(PC, 4)</code> value of the instruction to this label. Permitted values of the offset are -4095 to 4095. If the offset is zero or positive, <code>imm32</code> is equal to the offset and <code>add == TRUE</code> , encoded as <code>U == 1</code> . If the offset is negative, <code>imm32</code> is equal to minus the offset and <code>add == FALSE</code> , encoded as <code>U == 0</code> . |
| +/-     | Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in the "U" field. It can have the following values:<br>- when <code>U = 0</code><br>+ when <code>U = 1</code>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                  |
| <imm>   | Is a 12-bit unsigned immediate byte offset, in the range 0 to 4095, encoded in the "imm12" field.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                            |

### Operation for all encodings

```
1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 base = Align(PC, 4);
4 address = if add then (base + imm32) else (base - imm32);
5 data = MemU(address, 4);
6 if t == 15 then
7 if address[1:0] == '00' then
8 LoadWritePC(data, 0, Zeros(32), FALSE, FALSE);
9 else
10 UNPREDICTABLE;
11 else
12 R[t] = data;
```

### CONSTRAINED UNPREDICTABLE behavior

If `t == 15` && `address<1:0> != '00'`, then one of the following behaviors must occur:

- The instruction executes as described, with no change to its behavior and no additional side effects.
- The instruction generates an UNALIGNED UsageFault.

### C2.4.55 LDR (register)

Load Register (register). Load Register (register) calculates an address from a base register value and an offset register value, loads a word from memory, and writes it to a register. The offset register value can be shifted left by 0, 1, 2, or 3 bits.

The register loaded can be the PC. If it is, the word loaded for the PC is treated as a branch address or an exception return value. Bit[0] complies with the Arm architecture interworking rules for switching between the A32 and T32 instruction sets. However, Armv8-M only supports the T32 instruction set, so bit[0] must be 1. If bit[0] is 0 the PE takes an INVSTATE UsageFault exception on the instruction at the target address.

#### T1

Armv8-M

|    |    |    |    |    |    |   |   |    |   |   |   |    |   |    |   |
|----|----|----|----|----|----|---|---|----|---|---|---|----|---|----|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7  | 6 | 5 | 4 | 3  | 2 | 1  | 0 |
| 0  | 1  | 0  | 1  | 1  | 0  | 0 |   | Rm |   |   |   | Rn |   | Rt |   |

#### T1 variant

LDR{<c>}{<q>} <Rt>, [<Rn>, {+}<Rm>]

#### Decode for this encoding

```
1 t = UInt(Rt); n = UInt(Rn); m = UInt(Rm);
2 index = TRUE; add = TRUE; wback = FALSE;
3 (shift_t, shift_n) = (SRTYPE_LSL, 0);
```

#### T2

Armv8-M Main Extension only

|    |    |    |    |    |    |   |   |   |   |   |   |            |   |   |   |    |    |    |    |    |    |   |   |   |   |      |   |   |   |    |   |  |  |
|----|----|----|----|----|----|---|---|---|---|---|---|------------|---|---|---|----|----|----|----|----|----|---|---|---|---|------|---|---|---|----|---|--|--|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3          | 2 | 1 | 0 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5    | 4 | 3 | 2 | 1  | 0 |  |  |
| 1  | 1  | 1  | 1  | 1  | 0  | 0 | 0 | 0 | 1 | 0 | 1 | Rn != 1111 |   |   |   | Rt |    |    |    | 0  | 0  | 0 | 0 | 0 | 0 | imm2 |   |   |   | Rm |   |  |  |

#### T2 variant

LDR{<c>}.W <Rt>, [<Rn>, {+}<Rm>]  
 // <Rt>, <Rn>, <Rm> can be represented in T1  
 LDR{<c>}{<q>} <Rt>, [<Rn>, {+}<Rm>{, LSL #<imm>}]

#### Decode for this encoding

```
1 if Rn == '1111' then SEE "LDR (literal)";
2 if !HaveMainExt() then UNDEFINED;
3 t = UInt(Rt); n = UInt(Rn); m = UInt(Rm);
4 index = TRUE; add = TRUE; wback = FALSE;
5 (shift_t, shift_n) = (SRTYPE_LSL, UInt(imm2));
6 if m IN {13,15} then UNPREDICTABLE;
7 if t == 15 && InITBlock() && !LastInITBlock() then UNPREDICTABLE;
```

#### Assembler symbols for all encodings

- <c> See C1.2.5 *Standard assembler syntax fields* on page 309.
- <q> See C1.2.5 *Standard assembler syntax fields* on page 309.
- <Rt> For encoding T1: is the general-purpose register to be transferred, encoded in the "Rt" field.  
 For encoding T2: is the general-purpose register to be transferred, encoded in the "Rt" field.  
 The SP can be used. The PC can be used, provided the instruction is either outside an IT block or the last instruction of an IT block. If the PC is used, the instruction branches to the address (data) loaded to the PC.

|       |                                                                                                                                                                                   |
|-------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <Rn>  | Is the general-purpose base register, encoded in the "Rn" field.                                                                                                                  |
| +     | Specifies the index register is added to the base register.                                                                                                                       |
| <Rm>  | Is the general-purpose index register, encoded in the "Rm" field.                                                                                                                 |
| <imm> | If present, the size of the left shift to apply to the value from <Rm>, in the range 1-3. <imm> is encoded in imm2. If absent, no shift is specified and imm2 is encoded as 0b00. |

### Operation for all encodings

```
1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 offset = Shift(R[m], shift_t, shift_n, APSR.C);
4 offset_addr = if add then (R[n] + offset) else (R[n] - offset);
5 address = if index then offset_addr else R[n];
6
7 // Determine if the stack pointer limit should be checked
8 if n == 13 && wback then
9 (limit, applylimit) = LookUpSPLim(LookUpSP());
10 else
11 applylimit = FALSE;
12 // Memory operation only performed if limit not violated
13 if !applylimit || (UInt(offset_addr) >= UInt(limit)) then
14 data = MemU[address,4];
15
16 // If the stack pointer is being updated a fault will be raised if
17 // the limit is violated
18 if t == 15 then
19 if address[1:0] == '00' then
20 LoadWritePC(data, n, offset_addr, wback, TRUE);
21 else
22 UNPREDICTABLE;
23 else
24 if wback then RSPCheck[n] = offset_addr;
25 R[t] = data;
```

### CONSTRAINED UNPREDICTABLE behavior

If `t == 15 && address<1:0> != '00'`, then one of the following behaviors must occur:

- The instruction executes as described, with no change to its behavior and no additional side effects.
- The instruction generates an UNALIGNED UsageFault.

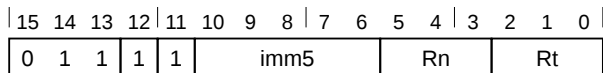


### C2.4.56 LDRB (immediate)

Load Register Byte (immediate). Load Register Byte (immediate) calculates an address from a base register value and an immediate offset, loads a byte from memory, zero-extends it to form a 32-bit word, and writes it to a register. It can use offset, post-indexed, or pre-indexed addressing.

#### T1

Armv8-M



#### T1 variant

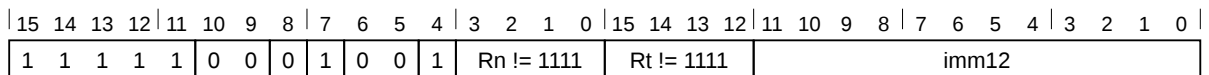
LDRB{<c>}{<q>} <Rt>, [<Rn> {, #<+><imm>}]

#### Decode for this encoding

```
1 t = UInt(Rt); n = UInt(Rn); imm32 = ZeroExtend(imm5, 32);
2 index = TRUE; add = TRUE; wback = FALSE;
```

#### T2

Armv8-M Main Extension only



#### T2 variant

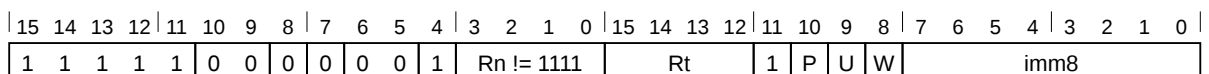
LDRB{<c>}.W <Rt>, [<Rn> {, #<+><imm>}]  
 // <Rt>, <Rn>, <imm> can be represented in T1  
 LDRB{<c>}{<q>} <Rt>, [<Rn> {, #<+><imm>}]

#### Decode for this encoding

```
1 if Rt == '1111' then SEE "PLD (immediate)";
2 if Rn == '1111' then SEE "LDRB (literal)";
3 if !HaveMainExt() then UNDEFINED;
4 t = UInt(Rt); n = UInt(Rn); imm32 = ZeroExtend(imm12, 32);
5 index = TRUE; add = TRUE; wback = FALSE;
6 if t == 13 then UNPREDICTABLE;
```

#### T3

Armv8-M Main Extension only



#### Offset variant

Applies when **Rt != 1111 && P == 1 && U == 0 && W == 0.**

LDRB{<c>}{<q>} <Rt>, [<Rn> {, #-<imm>}]

#### Post-indexed variant

Applies when **P == 0 && W == 1.**

LDRB{<c>}{<q>} <Rt>, [<Rn>], #{+/-}<imm>

### Pre-indexed variant

Applies when **P == 1 && W == 1**.

LDRB{<c>}{<q>} <Rt>, [<Rn>, #{+/-}<imm>]!

### Decode for this encoding

```

1 if Rt == '1111' && P == '1' && U == '0' && W == '0' then SEE "PLD (immediate)";
2 if Rn == '1111' then SEE "LDRB (literal)";
3 if P == '1' && U == '1' && W == '0' then SEE LDRBT;
4 if P == '0' && W == '0' then UNDEFINED;
5 if !HaveMainExt() then UNDEFINED;
6 t = UInt(Rt); n = UInt(Rn); imm32 = ZeroExtend(imm8, 32);
7 index = (P == '1'); add = (U == '1'); wback = (W == '1');
8 if t == 13 || (wback && n == t) then UNPREDICTABLE;
9 if t == 15 && W == '1' then UNPREDICTABLE;

```

### CONSTRAINED UNPREDICTABLE behavior

If `wback && n == t`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such an instruction, the base address might be corrupted so that the instruction cannot be repeated.

### Assembler symbols for all encodings

|       |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                   |
|-------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <c>   | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                          |
| <q>   | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                          |
| <Rt>  | Is the general-purpose register to be transferred, encoded in the "Rt" field.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                     |
| <Rn>  | For encoding T1: is the general-purpose base register, encoded in the "Rn" field.<br>For encoding T2 and T3: is the general-purpose base register, encoded in the "Rn" field. For PC use see <a href="#">C2.4.57 LDRB (literal)</a> on page 464.                                                                                                                                                                                                                                                                                                                                                  |
| +/-   | Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in the "U" field. It can have the following values:<br>- when U = 0<br>+ when U = 1                                                                                                                                                                                                                                                                                                                                                                                                 |
| +     | Specifies the offset is added to the base register.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                               |
| <imm> | For the post-indexed or pre-indexed variant: is an 8-bit unsigned immediate byte offset, in the range 0 to 255, encoded in the "imm8" field.<br>For encoding T1: is an optional 5-bit unsigned immediate byte offset, in the range 0 to 31, defaulting to 0 and encoded in the "imm5" field.<br>For encoding T2: is an optional 12-bit unsigned immediate byte offset, in the range 0 to 4095, defaulting to 0 and encoded in the "imm12" field.<br>For encoding T3: is an optional 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 and encoded in the "imm8" field. |

### Operation for all encodings

```

1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 offset_addr = if add then (R[n] + imm32) else (R[n] - imm32);
4 address = if index then offset_addr else R[n];

```

```
5
6 // Determine if the stack pointer limit should be checked
7 if n == 13 && wback then
8 (limit, applylimit) = LookupSPLim(LookupSP());
9 else
10 applylimit = FALSE;
11 // Memory operation only performed if limit not violated
12 if !applylimit || (UInt(offset_addr) >= UInt(limit)) then
13 R[t] = ZeroExtend(MemU[address,1], 32);
14
15 // If the stack pointer is being updated a fault will be raised if
16 // the limit is violated
17 if wback then RSPCheck[n] = offset_addr;
```

## C2.4.57 LDRB (literal)

Load Register Byte (literal). Load Register Byte (literal) calculates an address from the PC value and an immediate offset, loads a byte from memory, zero-extends it to form a 32-bit word, and writes it to a register.

### T1

Armv8-M Main Extension only

|    |    |    |    |    |    |   |   |   |   |   |   |   |   |   |   |            |    |    |    |    |    |   |   |   |   |   |       |   |   |   |   |  |
|----|----|----|----|----|----|---|---|---|---|---|---|---|---|---|---|------------|----|----|----|----|----|---|---|---|---|---|-------|---|---|---|---|--|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | 15         | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4     | 3 | 2 | 1 | 0 |  |
| 1  | 1  | 1  | 1  | 1  | 0  | 0 | 0 | U | 0 | 0 | 1 | 1 | 1 | 1 | 1 | Rt != 1111 |    |    |    |    |    |   |   |   |   |   | imm12 |   |   |   |   |  |

### T1 variant

```
LDRB{<c>}{<q>} <Rt>, <label>
// Preferred syntax
LDRB{<c>}{<q>} <Rt>, [PC, #{+/-}<imm>]
// Alternative syntax
```

### Decode for this encoding

```
1 if Rt == '1111' then SEE "PLD (literal)";
2 if !HaveMainExt() then UNDEFINED;
3 t = UInt(Rt); imm32 = ZeroExtend(imm12, 32); add = (U == '1');
4 if t == 13 then UNPREDICTABLE;
```

### Assembler symbols for all encodings

|         |                                                                                                                                                                                                                                                                                                                                                                                                                                                                      |
|---------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <c>     | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                                                                                                                                                                                                                                                                                                                                                                                             |
| <q>     | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                                                                                                                                                                                                                                                                                                                                                                                             |
| <Rt>    | Is the general-purpose register to be transferred, encoded in the "Rt" field.                                                                                                                                                                                                                                                                                                                                                                                        |
| <label> | The label of the literal data item that is to be loaded into <Rt>. The assembler calculates the required value of the offset from the <a href="#">Align</a> (PC, 4) value of the instruction to this label. Permitted values of the offset are -4095 to 4095. If the offset is zero or positive, imm32 is equal to the offset and add == TRUE, encoded as U == 1. If the offset is negative, imm32 is equal to minus the offset and add == FALSE, encoded as U == 0. |
| +/-     | Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in the "U" field. It can have the following values: <ul style="list-style-type: none"> <li>- when U = 0</li> <li>+ when U = 1</li> </ul>                                                                                                                                                                                                               |
| <imm>   | Is a 12-bit unsigned immediate byte offset, in the range 0 to 4095, encoded in the "imm12" field.                                                                                                                                                                                                                                                                                                                                                                    |

### Operation for all encodings

```
1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 base = Align(PC, 4);
4 address = if add then (base + imm32) else (base - imm32);
5 R[t] = ZeroExtend(MemU[address, 1], 32);
```

## C2.4.58 LDRB (register)

Load Register Byte (register). Load Register Byte (register) calculates an address from a base register value and an offset register value, loads a byte from memory, zero-extends it to form a 32-bit word, and writes it to a register. The offset register value can be shifted left by 0, 1, 2, or 3 bits.

### T1

Armv8-M

|    |    |    |    |    |    |   |   |    |   |    |   |   |   |   |    |
|----|----|----|----|----|----|---|---|----|---|----|---|---|---|---|----|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7  | 6 | 5  | 4 | 3 | 2 | 1 | 0  |
| 0  | 1  | 0  | 1  | 1  | 1  | 0 |   | Rm |   | Rn |   |   |   |   | Rt |

#### T1 variant

LDRB{<c>}{<q>} <Rt>, [<Rn>, {+}<Rm>]

#### Decode for this encoding

```
1 t = UInt(Rt); n = UInt(Rn); m = UInt(Rm);
2 index = TRUE; add = TRUE; wback = FALSE;
3 (shift_t, shift_n) = (SRTYPE_LSL, 0);
```

### T2

Armv8-M Main Extension only

|    |    |    |    |    |    |   |   |   |   |   |   |            |            |   |   |    |    |    |    |    |      |   |   |   |   |    |   |   |   |   |   |
|----|----|----|----|----|----|---|---|---|---|---|---|------------|------------|---|---|----|----|----|----|----|------|---|---|---|---|----|---|---|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3          | 2          | 1 | 0 | 15 | 14 | 13 | 12 | 11 | 10   | 9 | 8 | 7 | 6 | 5  | 4 | 3 | 2 | 1 | 0 |
| 1  | 1  | 1  | 1  | 1  | 0  | 0 | 0 | 0 | 0 | 0 | 1 | Rn != 1111 | Rt != 1111 | 0 | 0 | 0  | 0  | 0  | 0  | 0  | imm2 |   |   |   |   | Rm |   |   |   |   |   |

#### T2 variant

LDRB{<c>}.W <Rt>, [<Rn>, {+}<Rm>]  
 // <Rt>, <Rn>, <Rm> can be represented in T1  
 LDRB{<c>}{<q>} <Rt>, [<Rn>, {+}<Rm>{, LSL #<imm>}]

#### Decode for this encoding

```
1 if Rt == '1111' then SEE "PLD (register)";
2 if Rn == '1111' then SEE "LDRB (literal)";
3 if !HaveMainExt() then UNDEFINED;
4 t = UInt(Rt); n = UInt(Rn); m = UInt(Rm);
5 index = TRUE; add = TRUE; wback = FALSE;
6 (shift_t, shift_n) = (SRTYPE_LSL, UInt(imm2));
7 if t == 13 || m IN {13,15} then UNPREDICTABLE;
```

#### Assembler symbols for all encodings

|       |                                                                                                                                                                                   |
|-------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <c>   | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                                                                                                          |
| <q>   | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                                                                                                          |
| <Rt>  | Is the general-purpose register to be transferred, encoded in the "Rt" field.                                                                                                     |
| <Rn>  | Is the general-purpose base register, encoded in the "Rn" field.                                                                                                                  |
| +     | Specifies the index register is added to the base register.                                                                                                                       |
| <Rm>  | Is the general-purpose index register, encoded in the "Rm" field.                                                                                                                 |
| <imm> | If present, the size of the left shift to apply to the value from <Rm>, in the range 1-3. <imm> is encoded in imm2. If absent, no shift is specified and imm2 is encoded as 0b00. |

#### Operation for all encodings

```
1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 offset = Shift(R[m], shift_t, shift_n, APSR.C);
4 offset_addr = if add then (R[n] + offset) else (R[n] - offset);
5 address = if index then offset_addr else R[n];
6 R[t] = ZeroExtend(MemU[address,1],32);
```



### C2.4.60 LDRD (immediate)

Load Register Dual (immediate). Load Register Dual (immediate) calculates an address from a base register value and an immediate offset, loads two words from memory, and writes them to two registers. It can use offset, post-indexed, or pre-indexed addressing.

#### T1

Armv8-M Main Extension only

|    |    |    |    |    |    |   |   |   |   |   |   |            |   |   |   |    |    |    |    |     |    |   |   |      |   |   |   |   |   |   |   |
|----|----|----|----|----|----|---|---|---|---|---|---|------------|---|---|---|----|----|----|----|-----|----|---|---|------|---|---|---|---|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3          | 2 | 1 | 0 | 15 | 14 | 13 | 12 | 11  | 10 | 9 | 8 | 7    | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 1  | 1  | 1  | 0  | 1  | 0  | 0 | P | U | 1 | W | 1 | Rn != 1111 |   |   |   | Rt |    |    |    | Rt2 |    |   |   | imm8 |   |   |   |   |   |   |   |

#### Offset variant

Applies when **P == 1 && W == 0**.

LDRD{<c>}{<q>} <Rt>, <Rt2>, [<Rn> {, #+/-<imm>}]

#### Post-indexed variant

Applies when **P == 0 && W == 1**.

LDRD{<c>}{<q>} <Rt>, <Rt2>, [<Rn>], #+/-<imm>

#### Pre-indexed variant

Applies when **P == 1 && W == 1**.

LDRD{<c>}{<q>} <Rt>, <Rt2>, [<Rn>, #+/-<imm>]!

#### Decode for this encoding

```

1 if P == '0' && W == '0' then SEE "Related encodings";
2 if Rn == '1111' then SEE "LDRD (literal)";
3 if !HaveMainExt() then UNDEFINED;
4 t = UInt(Rt); t2 = UInt(Rt2); n = UInt(Rn); imm32 = ZeroExtend(imm8:'00', 32);
5 index = (P == '1'); add = (U == '1'); wback = (W == '1');
6 if wback && (n == t || n == t2) then UNPREDICTABLE;
7 if t IN {13,15} || t2 IN {13,15} || t == t2 then UNPREDICTABLE;
```

#### CONSTRAINED UNPREDICTABLE behavior

If `wback && (n == t || n == t2)`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such an instruction, the base address might be corrupted so that the instruction cannot be repeated.

#### CONSTRAINED UNPREDICTABLE behavior

If `t == t2`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The load instruction executes but the destination register takes an UNKNOWN value.



## Assembler symbols for all encodings

|       |                                                                                                                                                                                                                                                                                                                                                       |
|-------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <c>   | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                                                                                                                                                                                                                                                                              |
| <q>   | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                                                                                                                                                                                                                                                                              |
| <Rt>  | Is the first general-purpose register to be transferred, encoded in the "Rt" field.                                                                                                                                                                                                                                                                   |
| <Rt2> | Is the second general-purpose register to be transferred, encoded in the "Rt2" field.                                                                                                                                                                                                                                                                 |
| <Rn>  | Is the general-purpose base register, encoded in the "Rn" field. For PC use see <a href="#">C2.4.61 LDRD (literal)</a> on page 470.                                                                                                                                                                                                                   |
| +/-   | Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in the "U" field. It can have the following values: <ul style="list-style-type: none"><li>- when U = 0</li><li>+ when U = 1</li></ul>                                                                                                   |
| <imm> | For the offset variant: is the optional unsigned immediate byte offset, a multiple of 4, in the range 0 to 1020, defaulting to 0 and encoded in the "imm8" field as <imm>/4.<br>For the post-indexed and pre-indexed variant: is the unsigned immediate byte offset, a multiple of 4, in the range 0 to 1020, encoded in the "imm8" field as <imm>/4. |

## Operation for all encodings

```
1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 offset_addr = if add then (R[n] + imm32) else (R[n] - imm32);
4 address = if index then offset_addr else R[n];
5
6 // Determine if the stack pointer limit should be checked
7 if n == 13 && wback then
8 (limit, applylimit) = LookUpSPLim(LookUpSP());
9 else
10 applylimit = FALSE;
11 // Memory operation only performed if limit not violated
12 if !applylimit || (UInt(offset_addr) >= UInt(limit)) then
13 R[t] = MemA[address,4];
14 R[t2] = MemA[address+4,4];
15
16 // If the stack pointer is being updated a fault will be raised if
17 // the limit is violated
18 if wback then RSPCheck[n] = offset_addr;
```

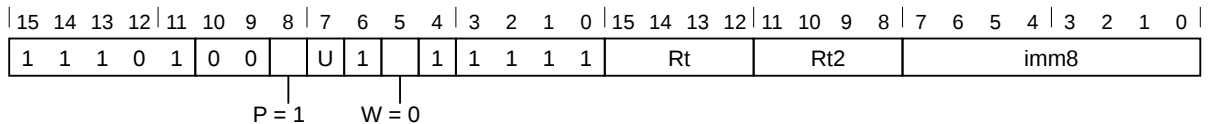
## C2.4.61 LDRD (literal)

Load Register Dual (literal). Load Register Dual (literal) calculates an address from the PC value and an immediate offset, loads two words from memory, and writes them to two registers.

For the M profile, the PC value must be word-aligned, otherwise the behavior of the instruction is UNPREDICTABLE.

### T1

Armv8-M Main Extension only



### T1 variant

```
LDRD{<c>}{<q>} <Rt>, <Rt2>, <label>
// Normal form
LDRD{<c>}{<q>} <Rt>, <Rt2>, [PC, # {+/-} <imm>]
// Alternative form
```

### Decode for this encoding

```
1 if P == '0' && W == '0' then SEE "Related encodings";
2 if P == '1' && W == '1' && U == '0' then SEE SG;
3 if !HaveMainExt() then UNDEFINED;
4 t = UInt(Rt); t2 = UInt(Rt2);
5 imm32 = ZeroExtend(imm8:'00', 32); add = (U == '1');
6 if t IN {13,15} || t2 IN {13,15} || t == t2 then UNPREDICTABLE;
7 if W == '1' then UNPREDICTABLE;
```

### CONSTRAINED UNPREDICTABLE behavior

If  $t == t2$ , then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The load instruction executes but the destination register takes an UNKNOWN value.

### CONSTRAINED UNPREDICTABLE behavior

If  $W == '1'$ , then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes without writeback of the base address.
- The instruction uses post-indexed addressing when  $P == '0'$  and uses pre-indexed addressing otherwise.

### Assembler symbols for all encodings

<c> See [C1.2.5 Standard assembler syntax fields](#) on page 309.  
 <q> See [C1.2.5 Standard assembler syntax fields](#) on page 309.  
 <Rt> Is the first general-purpose register to be transferred, encoded in the "Rt" field.  
 <Rt2> Is the second general-purpose register to be transferred, encoded in the "Rt2" field.

|         |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                  |
|---------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <label> | The label of the literal data item that is to be loaded into <Rt>. The assembler calculates the required value of the offset from the <a href="#">Align</a> (PC, 4) value of the instruction to this label. Permitted values of the offset are multiples of 4 in the range -1020 to 1020. If the offset is zero or positive, imm32 is equal to the offset and add == TRUE, encoded as U == 1. If the offset is negative, imm32 is equal to minus the offset and add == FALSE, encoded as U == 0. |
| +/-     | Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in the "U" field. It can have the following values: <ul style="list-style-type: none"><li>- when U = 0</li><li>+ when U = 1</li></ul>                                                                                                                                                                                                                                              |
| <imm>   | Is the optional 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 and encoded in the "imm8" field.                                                                                                                                                                                                                                                                                                                                                                    |

### Operation for all encodings

```
1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 if PC[1:0] != '00' then UNPREDICTABLE;
4 address = if add then (PC + imm32) else (PC - imm32);
5 R[t] = MemA[address,4];
6 R[t2] = MemA[address+4,4];
```

### CONSTRAINED UNPREDICTABLE behavior

If PC<1:0> != '00', then one of the following behaviors must occur:

- The instruction executes as described, with no change to its behavior and no additional side effects.
- The instruction generates an UNALIGNED UsageFault.

## C2.4.62 LDREX

Load Register Exclusive. Load Register Exclusive calculates an address from a base register value and an immediate offset, loads a word from memory, writes it to a register, and:

- If the address has the Shareable memory attribute, marks the physical address as exclusive access for the executing PE in a global monitor.
- Causes the executing PE to indicate an active exclusive access in the local monitor.

### T1

Armv8-M

|    |    |    |    |    |    |   |   |   |   |   |   |    |   |   |   |    |    |    |    |     |     |     |     |      |   |   |   |   |   |   |   |
|----|----|----|----|----|----|---|---|---|---|---|---|----|---|---|---|----|----|----|----|-----|-----|-----|-----|------|---|---|---|---|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3  | 2 | 1 | 0 | 15 | 14 | 13 | 12 | 11  | 10  | 9   | 8   | 7    | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 1  | 1  | 1  | 0  | 1  | 0  | 0 | 0 | 0 | 1 | 0 | 1 | Rn |   |   |   | Rt |    |    |    | (1) | (1) | (1) | (1) | imm8 |   |   |   |   |   |   |   |

### T1 variant

LDREX{<c>}{<q>} <Rt>, [<Rn> {, #<imm>}]

### Decode for this encoding

```
1 t = UInt(Rt); n = UInt(Rn); imm32 = ZeroExtend(imm8:'00', 32);
2 if t IN {13,15} || n == 15 then UNPREDICTABLE;
```

### Assembler symbols for all encodings

|       |                                                                                                                                                                        |
|-------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <c>   | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                                                                                               |
| <q>   | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                                                                                               |
| <Rt>  | Is the general-purpose register to be transferred, encoded in the "Rt" field.                                                                                          |
| <Rn>  | Is the general-purpose base register, encoded in the "Rn" field.                                                                                                       |
| <imm> | The immediate offset added to the value of <Rn> to calculate the address. <imm> can be omitted, meaning an offset of 0. Values are multiples of 4 in the range 0-1020. |

### Operation for all encodings

```
1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 address = R[n] + imm32;
4 SetExclusiveMonitors(address,4);
5 R[t] = MemA[address,4];
```

### C2.4.63 LDREXB

Load Register Exclusive Byte. Load Register Exclusive Byte derives an address from a base register value, loads a byte from memory, zero-extends it to form a 32-bit word, writes it to a register, and:

- If the address has the Shareable memory attribute, marks the physical address as exclusive access for the executing PE in a global monitor.
- Causes the executing PE to indicate an active exclusive access in the local monitor.

#### T1

Armv8-M

|    |    |    |    |    |    |   |   |   |   |   |   |   |   |   |    |    |    |    |    |    |     |     |     |     |   |   |   |   |     |     |     |     |
|----|----|----|----|----|----|---|---|---|---|---|---|---|---|---|----|----|----|----|----|----|-----|-----|-----|-----|---|---|---|---|-----|-----|-----|-----|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0  | 15 | 14 | 13 | 12 | 11 | 10  | 9   | 8   | 7   | 6 | 5 | 4 | 3 | 2   | 1   | 0   |     |
| 1  | 1  | 1  | 0  | 1  | 0  | 0 | 0 | 1 | 1 | 0 | 1 |   |   |   | Rn |    |    |    | Rt |    | (1) | (1) | (1) | (1) | 0 | 1 | 0 | 0 | (1) | (1) | (1) | (1) |

#### T1 variant

LDREXB{<c>}{<q>} <Rt>, [<Rn>]

#### Decode for this encoding

```
1 t = UInt(Rt); n = UInt(Rn);
2 if t IN {13,15} || n == 15 then UNPREDICTABLE;
```

#### Assembler symbols for all encodings

<c> See [C1.2.5 Standard assembler syntax fields](#) on page 309.  
 <q> See [C1.2.5 Standard assembler syntax fields](#) on page 309.  
 <Rt> Is the general-purpose register to be transferred, encoded in the "Rt" field.  
 <Rn> Is the general-purpose base register, encoded in the "Rn" field.

#### Operation for all encodings

```
1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 address = R[n];
4 SetExclusiveMonitors(address,1);
5 R[t] = ZeroExtend(MemA[address,1], 32);
```

## C2.4.64 LDREXH

Load Register Exclusive Halfword. Load Register Exclusive Halfword derives an address from a base register value, loads a halfword from memory, zero-extends it to form a 32-bit word, writes it to a register, and:

- If the address has the Shareable memory attribute, marks the physical address as exclusive access for the executing PE in a global monitor.
- Causes the executing PE to indicate an active exclusive access in the local monitor.

### T1

Armv8-M

|    |    |    |    |    |    |   |   |   |   |   |   |   |   |    |   |    |    |    |    |     |     |     |     |   |   |   |   |     |     |     |     |
|----|----|----|----|----|----|---|---|---|---|---|---|---|---|----|---|----|----|----|----|-----|-----|-----|-----|---|---|---|---|-----|-----|-----|-----|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1  | 0 | 15 | 14 | 13 | 12 | 11  | 10  | 9   | 8   | 7 | 6 | 5 | 4 | 3   | 2   | 1   | 0   |
| 1  | 1  | 1  | 0  | 1  | 0  | 0 | 0 | 1 | 1 | 0 | 1 |   |   | Rn |   |    | Rt |    |    | (1) | (1) | (1) | (1) | 0 | 1 | 0 | 1 | (1) | (1) | (1) | (1) |

### T1 variant

LDREXH{<c>}{<q>} <Rt>, [<Rn>]

### Decode for this encoding

```
1 t = UInt(Rt); n = UInt(Rn);
2 if t IN {13,15} || n == 15 then UNPREDICTABLE;
```

### Assembler symbols for all encodings

<c> See [C1.2.5 Standard assembler syntax fields](#) on page 309.  
 <q> See [C1.2.5 Standard assembler syntax fields](#) on page 309.  
 <Rt> Is the general-purpose register to be transferred, encoded in the "Rt" field.  
 <Rn> Is the general-purpose base register, encoded in the "Rn" field.

### Operation for all encodings

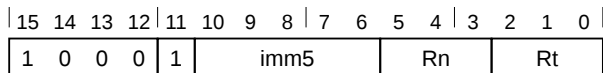
```
1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 address = R[n];
4 SetExclusiveMonitors(address,2);
5 R[t] = ZeroExtend(MemA[address,2], 32);
```

### C2.4.65 LDRH (immediate)

Load Register Halfword (immediate). Load Register Halfword (immediate) calculates an address from a base register value and an immediate offset, loads a halfword from memory, zero-extends it to form a 32-bit word, and writes it to a register. It can use offset, post-indexed, or pre-indexed addressing.

#### T1

Armv8-M



#### T1 variant

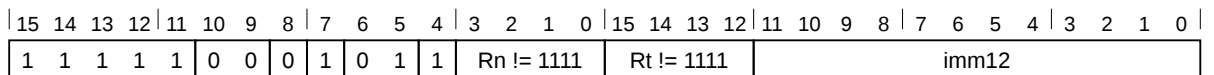
LDRH{<c>}{<q>} <Rt>, [<Rn> {, #<+><imm>}]

#### Decode for this encoding

```
1 t = UInt(Rt); n = UInt(Rn); imm32 = ZeroExtend(imm5:'0', 32);
2 index = TRUE; add = TRUE; wback = FALSE;
```

#### T2

Armv8-M Main Extension only



#### T2 variant

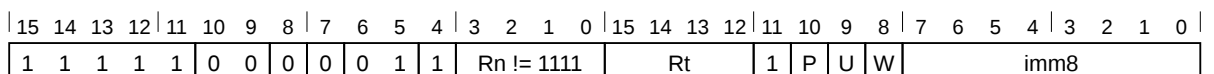
LDRH{<c>}.W <Rt>, [<Rn> {, #<+><imm>}]  
 // <Rt>, <Rn>, <imm> can be represented in T1  
 LDRH{<c>}{<q>} <Rt>, [<Rn> {, #<+><imm>}]

#### Decode for this encoding

```
1 if Rt == '1111' then SEE "Related encodings";
2 if Rn == '1111' then SEE "LDRH (literal)";
3 if !HaveMainExt() then UNDEFINED;
4 t = UInt(Rt); n = UInt(Rn); imm32 = ZeroExtend(imm12, 32);
5 index = TRUE; add = TRUE; wback = FALSE;
6 if t == 13 then UNPREDICTABLE;
```

#### T3

Armv8-M Main Extension only



#### Offset variant

Applies when **Rt != 1111 && P == 1 && U == 0 && W == 0.**

LDRH{<c>}{<q>} <Rt>, [<Rn> {, #-<imm>}]

#### Post-indexed variant

Applies when **P == 0 && W == 1.**

LDRH{<c>}{<q>} <Rt>, [<Rn>], #{+/-}<imm>

### Pre-indexed variant

Applies when **P == 1 && W == 1**.

LDRH{<c>}{<q>} <Rt>, [<Rn>, #{+/-}<imm>]!

### Decode for this encoding

```

1 if Rn == '1111' then SEE "LDRH (literal)";
2 if Rt == '1111' && P == '1' && U == '0' && W == '0' then SEE "Related encodings";
3 if P == '1' && U == '1' && W == '0' then SEE LDRHT;
4 if P == '0' && W == '0' then UNDEFINED;
5 if !HaveMainExt() then UNDEFINED;
6 t = UInt(Rt); n = UInt(Rn); imm32 = ZeroExtend(imm8, 32);
7 index = (P == '1'); add = (U == '1'); wback = (W == '1');
8 if t == 13 || (t == 15 && W == '1') || (wback && n == t) then UNPREDICTABLE;
```

### CONSTRAINED UNPREDICTABLE behavior

If **wback && n == t**, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such an instruction, the base address might be corrupted so that the instruction cannot be repeated.

### Assembler symbols for all encodings

|       |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                  |
|-------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <c>   | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |
| <q>   | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |
| <Rt>  | Is the general-purpose register to be transferred, encoded in the "Rt" field.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                    |
| <Rn>  | For encoding T1: is the general-purpose base register, encoded in the "Rn" field.<br>For encoding T2 and T3: is the general-purpose base register, encoded in the "Rn" field. For PC use see <a href="#">C2.4.66 LDRH (literal)</a> on page 478.                                                                                                                                                                                                                                                                                                                                                                                 |
| +/-   | Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in the "U" field. It can have the following values: <ul style="list-style-type: none"> <li>- when U = 0</li> <li>+ when U = 1</li> </ul>                                                                                                                                                                                                                                                                                                                                                                           |
| +     | Specifies the offset is added to the base register.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                              |
| <imm> | For the post-indexed or pre-indexed variant: is an 8-bit unsigned immediate byte offset, in the range 0 to 255, encoded in the "imm8" field.<br>For encoding T1: is the optional positive unsigned immediate byte offset, a multiple of 2 in the range 0 to 62, defaulting to 0 and encoded in the "imm5" field as <imm>/2.<br>For encoding T2: is an optional 12-bit unsigned immediate byte offset, in the range 0 to 4095, defaulting to 0 and encoded in the "imm12" field.<br>For encoding T3: is an optional 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 and encoded in the "imm8" field. |

### Operation for all encodings

```

1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 offset_addr = if add then (R[n] + imm32) else (R[n] - imm32);
4 address = if index then offset_addr else R[n];
5
```



```
6 // Determine if the stack pointer limit should be checked
7 if n == 13 && wback then
8 (limit, applylimit) = LookUpSPLim(LookUpSP());
9 else
10 applylimit = FALSE;
11 // Memory operation only performed if limit not violated
12 if !applylimit || (UInt(offset_addr) >= UInt(limit)) then
13 R[t] = ZeroExtend(MemU[address,2], 32);
14
15 // If the stack pointer is being updated a fault will be raised if
16 // the limit is violated
17 if wback then RSPCheck[n] = offset_addr;
```

## C2.4.66 LDRH (literal)

Load Register Halfword (literal). Load Register Halfword (literal) calculates an address from the PC value and an immediate offset, loads a halfword from memory, zero-extends it to form a 32-bit word, and writes it to a register.

### T1

Armv8-M Main Extension only

|    |    |    |    |    |    |   |   |   |   |   |   |   |   |   |   |            |    |    |    |    |    |   |   |   |   |   |       |   |   |   |   |  |
|----|----|----|----|----|----|---|---|---|---|---|---|---|---|---|---|------------|----|----|----|----|----|---|---|---|---|---|-------|---|---|---|---|--|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | 15         | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4     | 3 | 2 | 1 | 0 |  |
| 1  | 1  | 1  | 1  | 1  | 0  | 0 | 0 | U | 0 | 1 | 1 | 1 | 1 | 1 | 1 | Rt != 1111 |    |    |    |    |    |   |   |   |   |   | imm12 |   |   |   |   |  |

### T1 variant

```
LDRH{<c>}{<q>} <Rt>, <label>
// Preferred syntax
LDRH{<c>}{<q>} <Rt>, [PC, #{+/-}<imm>]
// Alternative syntax
```

### Decode for this encoding

```
1 if Rt == '1111' then SEE "PLD (literal)";
2 if !HaveMainExt() then UNDEFINED;
3 t = UInt(Rt); imm32 = ZeroExtend(imm12, 32); add = (U == '1');
4 if t == 13 then UNPREDICTABLE;
```

### Assembler symbols for all encodings

|         |                                                                                                                                                                                                                                                                                                                                                                                                                                                                      |
|---------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <c>     | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                                                                                                                                                                                                                                                                                                                                                                                             |
| <q>     | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                                                                                                                                                                                                                                                                                                                                                                                             |
| <Rt>    | Is the general-purpose register to be transferred, encoded in the "Rt" field.                                                                                                                                                                                                                                                                                                                                                                                        |
| <label> | The label of the literal data item that is to be loaded into <Rt>. The assembler calculates the required value of the offset from the <a href="#">Align</a> (PC, 4) value of the instruction to this label. Permitted values of the offset are -4095 to 4095. If the offset is zero or positive, imm32 is equal to the offset and add == TRUE, encoded as U == 1. If the offset is negative, imm32 is equal to minus the offset and add == FALSE, encoded as U == 0. |
| +/-     | Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in the "U" field. It can have the following values: <ul style="list-style-type: none"> <li>- when U = 0</li> <li>+ when U = 1</li> </ul>                                                                                                                                                                                                               |
| <imm>   | Is a 12-bit unsigned immediate byte offset, in the range 0 to 4095, encoded in the "imm12" field.                                                                                                                                                                                                                                                                                                                                                                    |

### Operation for all encodings

```
1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 base = Align(PC, 4);
4 address = if add then (base + imm32) else (base - imm32);
5 data = MemU[address, 2];
6 R[t] = ZeroExtend(data, 32);
```

### C2.4.67 LDRH (register)

Load Register Halfword (register). Load Register Halfword (register) calculates an address from a base register value and an offset register value, loads a halfword from memory, zero-extends it to form a 32-bit word, and writes it to a register. The offset register value can be shifted left by 0, 1, 2, or 3 bits.

#### T1

Armv8-M

|    |    |    |    |    |    |   |   |    |   |    |   |   |   |   |    |
|----|----|----|----|----|----|---|---|----|---|----|---|---|---|---|----|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7  | 6 | 5  | 4 | 3 | 2 | 1 | 0  |
| 0  | 1  | 0  | 1  | 1  | 0  | 1 |   | Rm |   | Rn |   |   |   |   | Rt |

#### T1 variant

LDRH{<c>}{<q>} <Rt>, [<Rn>, {+}<Rm>]

#### Decode for this encoding

```
1 t = UInt(Rt); n = UInt(Rn); m = UInt(Rm);
2 index = TRUE; add = TRUE; wback = FALSE;
3 (shift_t, shift_n) = (SRTYPE_LSL, 0);
```

#### T2

Armv8-M Main Extension only

|    |    |    |    |    |    |   |   |   |   |   |   |            |            |   |   |    |    |    |    |    |      |   |   |   |   |   |    |   |   |   |   |
|----|----|----|----|----|----|---|---|---|---|---|---|------------|------------|---|---|----|----|----|----|----|------|---|---|---|---|---|----|---|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3          | 2          | 1 | 0 | 15 | 14 | 13 | 12 | 11 | 10   | 9 | 8 | 7 | 6 | 5 | 4  | 3 | 2 | 1 | 0 |
| 1  | 1  | 1  | 1  | 1  | 0  | 0 | 0 | 0 | 0 | 1 | 1 | Rn != 1111 | Rt != 1111 | 0 | 0 | 0  | 0  | 0  | 0  | 0  | imm2 |   |   |   |   |   | Rm |   |   |   |   |

#### T2 variant

LDRH{<c>}.W <Rt>, [<Rn>, {+}<Rm>]  
 // <Rt>, <Rn>, <Rm> can be represented in T1  
 LDRH{<c>}{<q>} <Rt>, [<Rn>, {+}<Rm>{, LSL #<imm>}]

#### Decode for this encoding

```
1 if Rn == '1111' then SEE "LDRH (literal)";
2 if Rt == '1111' then SEE "Related encodings";
3 if !HaveMainExt() then UNDEFINED;
4 t = UInt(Rt); n = UInt(Rn); m = UInt(Rm);
5 index = TRUE; add = TRUE; wback = FALSE;
6 (shift_t, shift_n) = (SRTYPE_LSL, UInt(imm2));
7 if t == 13 || m IN {13,15} then UNPREDICTABLE;
```

#### Assembler symbols for all encodings

|       |                                                                                                                                                                                   |
|-------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <c>   | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                                                                                                          |
| <q>   | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                                                                                                          |
| <Rt>  | Is the general-purpose register to be transferred, encoded in the "Rt" field.                                                                                                     |
| <Rn>  | Is the general-purpose base register, encoded in the "Rn" field.                                                                                                                  |
| +     | Specifies the index register is added to the base register.                                                                                                                       |
| <Rm>  | Is the general-purpose index register, encoded in the "Rm" field.                                                                                                                 |
| <imm> | If present, the size of the left shift to apply to the value from <Rm>, in the range 1-3. <imm> is encoded in imm2. If absent, no shift is specified and imm2 is encoded as 0b00. |

#### Operation for all encodings

```
1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 offset = Shift(R[m], shift_t, shift_n, APSR.C);
4 offset_addr = if add then (R[n] + offset) else (R[n] - offset);
5 address = if index then offset_addr else R[n];
6 data = MemU(address, 2);
7 if wback then R[n] = offset_addr;
8 R[t] = ZeroExtend(data, 32);
```

## C2.4.68 LDRHT

Load Register Halfword Unprivileged. Load Register Halfword Unprivileged calculates an address from a base register value and an immediate offset, loads a halfword from memory, zero-extends it to form a 32-bit word, and writes it to a register.

When privileged software uses an **LDRHT** instruction, the memory access is restricted as if the software was unprivileged.

### T1

*Armv8-M Main Extension only*

|    |    |    |    |    |    |   |   |   |   |   |   |            |   |   |   |    |    |    |    |    |    |   |   |      |   |   |   |   |   |   |   |
|----|----|----|----|----|----|---|---|---|---|---|---|------------|---|---|---|----|----|----|----|----|----|---|---|------|---|---|---|---|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3          | 2 | 1 | 0 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7    | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 1  | 1  | 1  | 1  | 1  | 0  | 0 | 0 | 0 | 0 | 1 | 1 | Rn != 1111 |   |   |   | Rt |    |    |    | 1  | 1  | 1 | 0 | imm8 |   |   |   |   |   |   |   |

### T1 variant

LDRHT{<c>}{<q>} <Rt>, [<Rn> {, #<+><imm>}]

### Decode for this encoding

```

1 if Rn == '1111' then SEE "LDRH (literal)";
2 if !HaveMainExt() then UNDEFINED;
3 t = UInt(Rt); n = UInt(Rn); postindex = FALSE; add = TRUE;
4 register_form = FALSE; imm32 = ZeroExtend(imm8, 32);
5 if t IN {13,15} then UNPREDICTABLE;

```

### Assembler symbols for all encodings

|       |                                                                                                                              |
|-------|------------------------------------------------------------------------------------------------------------------------------|
| <c>   | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                                                     |
| <q>   | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                                                     |
| <Rt>  | Is the general-purpose register to be transferred, encoded in the "Rt" field.                                                |
| <Rn>  | Is the general-purpose base register, encoded in the "Rn" field.                                                             |
| +     | Specifies the offset is added to the base register.                                                                          |
| <imm> | Is an optional 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 and encoded in the "imm8" field. |

### Operation for all encodings

```

1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 address = R[n] + imm32;
4 data = MemU_unpriv(address,2);
5 R[t] = ZeroExtend(data, 32);

```

## C2.4.69 LDRSB (immediate)

Load Register Signed Byte (immediate). Load Register Signed Byte (immediate) calculates an address from a base register value and an immediate offset, loads a byte from memory, sign-extends it to form a 32-bit word, and writes it to a register. It can use offset, post-indexed, or pre-indexed addressing.

### T1

Armv8-M Main Extension only

|    |    |    |    |    |    |   |   |   |   |   |   |            |            |       |   |    |    |    |    |    |    |   |   |   |   |   |   |   |   |   |   |
|----|----|----|----|----|----|---|---|---|---|---|---|------------|------------|-------|---|----|----|----|----|----|----|---|---|---|---|---|---|---|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3          | 2          | 1     | 0 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 1  | 1  | 1  | 1  | 1  | 0  | 0 | 1 | 1 | 0 | 0 | 1 | Rn != 1111 | Rt != 1111 | imm12 |   |    |    |    |    |    |    |   |   |   |   |   |   |   |   |   |   |

### T1 variant

LDRSB{<c>}{<q>} <Rt>, [<Rn> {, #+}<imm>]}

### Decode for this encoding

```

1 if Rt == '1111' then SEE "PLI (immediate, literal)";
2 if Rn == '1111' then SEE "LDRSB (literal)";
3 if !HaveMainExt() then UNDEFINED;
4 t = UInt(Rt); n = UInt(Rn); imm32 = ZeroExtend(imm12, 32);
5 index = TRUE; add = TRUE; wback = FALSE;
6 if t == 13 then UNPREDICTABLE;

```

### T2

Armv8-M Main Extension only

|    |    |    |    |    |    |   |   |   |   |   |   |            |    |   |   |    |    |      |    |    |    |   |   |   |   |   |   |   |   |   |   |
|----|----|----|----|----|----|---|---|---|---|---|---|------------|----|---|---|----|----|------|----|----|----|---|---|---|---|---|---|---|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3          | 2  | 1 | 0 | 15 | 14 | 13   | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 1  | 1  | 1  | 1  | 1  | 0  | 0 | 1 | 0 | 0 | 0 | 1 | Rn != 1111 | Rt | 1 | P | U  | W  | imm8 |    |    |    |   |   |   |   |   |   |   |   |   |   |

### Offset variant

Applies when **P == 1 && U == 0 && W == 0**.

LDRSB{<c>}{<q>} <Rt>, [<Rn> {, #-}<imm>]}

### Post-indexed variant

Applies when **P == 0 && W == 1**.

LDRSB{<c>}{<q>} <Rt>, [<Rn>], #{+/-}<imm>

### Pre-indexed variant

Applies when **P == 1 && W == 1**.

LDRSB{<c>}{<q>} <Rt>, [<Rn>], #{+/-}<imm>]!

### Decode for this encoding

```

1 if Rt == '1111' && P == '1' && U == '0' && W == '0' then SEE "PLI (immediate, literal)";
2 if Rn == '1111' then SEE "LDRSB (literal)";
3 if P == '1' && U == '1' && W == '0' then SEE LDRSBT;
4 if P == '0' && W == '0' then UNDEFINED;
5 if !HaveMainExt() then UNDEFINED;
6 t = UInt(Rt); n = UInt(Rn); imm32 = ZeroExtend(imm8, 32);
7 index = (P == '1'); add = (U == '1'); wback = (W == '1');
8 if t == 13 || (t == 15 && W == '1') || (wback && n == t) then UNPREDICTABLE;

```

## CONSTRAINED UNPREDICTABLE behavior

If `wback && n == t`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such an instruction, the base address might be corrupted so that the instruction cannot be repeated.

## Assembler symbols for all encodings

|       |                                                                                                                                                                                                                                                                                                                                                                                                                                                   |
|-------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <c>   | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                                                                                                                                                                                                                                                                                                                                                                          |
| <q>   | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                                                                                                                                                                                                                                                                                                                                                                          |
| <Rt>  | Is the general-purpose register to be transferred, encoded in the "Rt" field.                                                                                                                                                                                                                                                                                                                                                                     |
| <Rn>  | Is the general-purpose base register, encoded in the "Rn" field. For PC use see <a href="#">C2.4.70 LDRSB (literal)</a> on page 484.                                                                                                                                                                                                                                                                                                              |
| +/-   | Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in the "U" field. It can have the following values: <ul style="list-style-type: none"><li>- when U = 0</li><li>+ when U = 1</li></ul>                                                                                                                                                                                               |
| +     | Specifies the offset is added to the base register.                                                                                                                                                                                                                                                                                                                                                                                               |
| <imm> | For the post-indexed or pre-indexed variant: is an 8-bit unsigned immediate byte offset, in the range 0 to 255, encoded in the "imm8" field.<br>For encoding T1: is an optional 12-bit unsigned immediate byte offset, in the range 0 to 4095, defaulting to 0 and encoded in the "imm12" field.<br>For encoding T2: is an optional 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 and encoded in the "imm8" field. |

## Operation for all encodings

```
1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 offset_addr = if add then (R[n] + imm32) else (R[n] - imm32);
4 address = if index then offset_addr else R[n];
5
6 // Determine if the stack pointer limit should be checked
7 if n == 13 && wback then
8 (limit, applylimit) = LookUpSPLim(LookUpSP());
9 else
10 applylimit = FALSE;
11 // Memory operation only performed if limit not violated
12 if !applylimit || (UInt(offset_addr) >= UInt(limit)) then
13 R[t] = SignExtend(MemU[address,1], 32);
14
15 // If the stack pointer is being updated a fault will be raised if
16 // the limit is violated
17 if wback then RSPCheck[n] = offset_addr;
```

## C2.4.70 LDRSB (literal)

Load Register Signed Byte (literal). Load Register Signed Byte (literal) calculates an address from the PC value and an immediate offset, loads a byte from memory, sign-extends it to form a 32-bit word, and writes it to a register.

### T1

Armv8-M Main Extension only

|    |    |    |    |    |    |   |   |   |   |   |   |   |   |   |   |            |    |    |    |    |    |   |   |   |   |   |       |   |   |   |   |  |
|----|----|----|----|----|----|---|---|---|---|---|---|---|---|---|---|------------|----|----|----|----|----|---|---|---|---|---|-------|---|---|---|---|--|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | 15         | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4     | 3 | 2 | 1 | 0 |  |
| 1  | 1  | 1  | 1  | 1  | 0  | 0 | 1 | U | 0 | 0 | 1 | 1 | 1 | 1 | 1 | Rt != 1111 |    |    |    |    |    |   |   |   |   |   | imm12 |   |   |   |   |  |

### T1 variant

```
LDRSB{<c>}{<q>} <Rt>, <label>
 // Preferred syntax
LDRSB{<c>}{<q>} <Rt>, [PC, #{+/-}<imm>]
 // Alternative syntax
```

### Decode for this encoding

```
1 if Rt == '1111' then SEE "PLI (immediate, literal)";
2 if !HaveMainExt() then UNDEFINED;
3 t = UInt(Rt); imm32 = ZeroExtend(imm12, 32); add = (U == '1');
4 if t == 13 then UNPREDICTABLE;
```

### Assembler symbols for all encodings

|         |                                                                                                                                                                                                                                                                                                                                                                                                                                                                      |
|---------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <c>     | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                                                                                                                                                                                                                                                                                                                                                                                             |
| <q>     | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                                                                                                                                                                                                                                                                                                                                                                                             |
| <Rt>    | Is the general-purpose register to be transferred, encoded in the "Rt" field.                                                                                                                                                                                                                                                                                                                                                                                        |
| <label> | The label of the literal data item that is to be loaded into <Rt>. The assembler calculates the required value of the offset from the <a href="#">Align</a> (PC, 4) value of the instruction to this label. Permitted values of the offset are -4095 to 4095. If the offset is zero or positive, imm32 is equal to the offset and add == TRUE, encoded as U == 1. If the offset is negative, imm32 is equal to minus the offset and add == FALSE, encoded as U == 0. |
| +/-     | Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in the "U" field. It can have the following values: <ul style="list-style-type: none"> <li>- when U = 0</li> <li>+ when U = 1</li> </ul>                                                                                                                                                                                                               |
| <imm>   | Is a 12-bit unsigned immediate byte offset, in the range 0 to 4095, encoded in the "imm12" field.                                                                                                                                                                                                                                                                                                                                                                    |

### Operation for all encodings

```
1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 base = Align(PC, 4);
4 address = if add then (base + imm32) else (base - imm32);
5 R[t] = SignExtend(MemU[address, 1], 32);
```



### C2.4.71 LDRSB (register)

Load Register Signed Byte (register). Load Register Signed Byte (register) calculates an address from a base register value and an offset register value, loads a byte from memory, sign-extends it to form a 32-bit word, and writes it to a register. The offset register value can be shifted left by 0, 1, 2, or 3 bits.

#### T1

Armv8-M

|    |    |    |    |    |    |   |   |    |   |    |   |   |   |   |    |
|----|----|----|----|----|----|---|---|----|---|----|---|---|---|---|----|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7  | 6 | 5  | 4 | 3 | 2 | 1 | 0  |
| 0  | 1  | 0  | 1  | 0  | 1  | 1 |   | Rm |   | Rn |   |   |   |   | Rt |

#### T1 variant

LDRSB{<c>}{<q>} <Rt>, [<Rn>, {+}<Rm>]

#### Decode for this encoding

```
1 t = UInt(Rt); n = UInt(Rn); m = UInt(Rm);
2 index = TRUE; add = TRUE; wback = FALSE;
3 (shift_t, shift_n) = (SRTYPE_LSL, 0);
```

#### T2

Armv8-M Main Extension only

|    |    |    |    |    |    |   |   |   |   |   |   |            |            |   |   |    |    |    |    |    |      |   |   |   |   |   |    |   |   |   |   |
|----|----|----|----|----|----|---|---|---|---|---|---|------------|------------|---|---|----|----|----|----|----|------|---|---|---|---|---|----|---|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3          | 2          | 1 | 0 | 15 | 14 | 13 | 12 | 11 | 10   | 9 | 8 | 7 | 6 | 5 | 4  | 3 | 2 | 1 | 0 |
| 1  | 1  | 1  | 1  | 1  | 0  | 0 | 1 | 0 | 0 | 0 | 1 | Rn != 1111 | Rt != 1111 | 0 | 0 | 0  | 0  | 0  | 0  | 0  | imm2 |   |   |   |   |   | Rm |   |   |   |   |

#### T2 variant

LDRSB{<c>}.W <Rt>, [<Rn>, {+}<Rm>]  
 // <Rt>, <Rn>, <Rm> can be represented in T1  
 LDRSB{<c>}{<q>} <Rt>, [<Rn>, {+}<Rm>{, LSL #<imm>}]

#### Decode for this encoding

```
1 if Rt == '1111' then SEE "PLI (register)";
2 if Rn == '1111' then SEE "LDRSB (literal)";
3 if !HaveMainExt() then UNDEFINED;
4 t = UInt(Rt); n = UInt(Rn); m = UInt(Rm);
5 index = TRUE; add = TRUE; wback = FALSE;
6 (shift_t, shift_n) = (SRTYPE_LSL, UInt(imm2));
7 if t == 13 || m IN {13,15} then UNPREDICTABLE;
```

#### Assembler symbols for all encodings

|       |                                                                                                                                                                                   |
|-------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <c>   | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                                                                                                          |
| <q>   | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                                                                                                          |
| <Rt>  | Is the general-purpose register to be transferred, encoded in the "Rt" field.                                                                                                     |
| <Rn>  | Is the general-purpose base register, encoded in the "Rn" field.                                                                                                                  |
| +     | Specifies the index register is added to the base register.                                                                                                                       |
| <Rm>  | Is the general-purpose index register, encoded in the "Rm" field.                                                                                                                 |
| <imm> | If present, the size of the left shift to apply to the value from <Rm>, in the range 1-3. <imm> is encoded in imm2. If absent, no shift is specified and imm2 is encoded as 0b00. |

#### Operation for all encodings

```
1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 offset = Shift(R[m], shift_t, shift_n, APSR.C);
4 offset_addr = if add then (R[n] + offset) else (R[n] - offset);
5 address = if index then offset_addr else R[n];
6 R[t] = SignExtend(MemU[address,1], 32);
```



### C2.4.73 LDRSH (immediate)

Load Register Signed Halfword (immediate). Load Register Signed Halfword (immediate) calculates an address from a base register value and an immediate offset, loads a halfword from memory, sign-extends it to form a 32-bit word, and writes it to a register. It can use offset, post-indexed, or pre-indexed addressing.

#### T1

Armv8-M Main Extension only

|    |    |    |    |    |    |   |   |   |   |   |   |            |            |       |   |    |    |    |    |    |    |   |   |   |   |   |   |   |   |   |   |
|----|----|----|----|----|----|---|---|---|---|---|---|------------|------------|-------|---|----|----|----|----|----|----|---|---|---|---|---|---|---|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3          | 2          | 1     | 0 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 1  | 1  | 1  | 1  | 1  | 0  | 0 | 1 | 1 | 0 | 1 | 1 | Rn != 1111 | Rt != 1111 | imm12 |   |    |    |    |    |    |    |   |   |   |   |   |   |   |   |   |   |

#### T1 variant

LDRSH{<c>}{<q>} <Rt>, [<Rn> {, #+<imm>}]

#### Decode for this encoding

```

1 if Rn == '1111' then SEE "LDRSH (literal)";
2 if Rt == '1111' then SEE "Related encodings";
3 if !HaveMainExt() then UNDEFINED;
4 t = UInt(Rt); n = UInt(Rn); imm32 = ZeroExtend(imm12, 32);
5 index = TRUE; add = TRUE; wback = FALSE;
6 if t == 13 then UNPREDICTABLE;

```

#### T2

Armv8-M Main Extension only

|    |    |    |    |    |    |   |   |   |   |   |   |            |    |   |   |    |    |      |    |    |    |   |   |   |   |   |   |   |   |   |   |
|----|----|----|----|----|----|---|---|---|---|---|---|------------|----|---|---|----|----|------|----|----|----|---|---|---|---|---|---|---|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3          | 2  | 1 | 0 | 15 | 14 | 13   | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 1  | 1  | 1  | 1  | 1  | 0  | 0 | 1 | 0 | 0 | 1 | 1 | Rn != 1111 | Rt | 1 | P | U  | W  | imm8 |    |    |    |   |   |   |   |   |   |   |   |   |   |

#### Offset variant

Applies when Rt != 1111 && P == 1 && U == 0 && W == 0.

LDRSH{<c>}{<q>} <Rt>, [<Rn> {, #-<imm>}]

#### Post-indexed variant

Applies when P == 0 && W == 1.

LDRSH{<c>}{<q>} <Rt>, [<Rn>], #{+/-}<imm>

#### Pre-indexed variant

Applies when P == 1 && W == 1.

LDRSH{<c>}{<q>} <Rt>, [<Rn>, #{+/-}<imm>]!

#### Decode for this encoding

```

1 if Rn == '1111' then SEE "LDRSH (literal)";
2 if Rt == '1111' && P == '1' && U == '0' && W == '0' then SEE "Related encodings";
3 if P == '1' && U == '1' && W == '0' then SEE LDRSHT;
4 if P == '0' && W == '0' then UNDEFINED;
5 if !HaveMainExt() then UNDEFINED;
6 t = UInt(Rt); n = UInt(Rn); imm32 = ZeroExtend(imm8, 32);
7 index = (P == '1'); add = (U == '1'); wback = (W == '1');
8 if t == 13 || (t == 15 && W == '1') || (wback && n == t) then UNPREDICTABLE;

```

## CONSTRAINED UNPREDICTABLE behavior

If `wback && n == t`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such an instruction, the base address might be corrupted so that the instruction cannot be repeated.

## Assembler symbols for all encodings

|       |                                                                                                                                                                                                                                                                                                                                                                                                                                                   |
|-------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <c>   | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                                                                                                                                                                                                                                                                                                                                                                          |
| <q>   | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                                                                                                                                                                                                                                                                                                                                                                          |
| <Rt>  | Is the general-purpose register to be transferred, encoded in the "Rt" field.                                                                                                                                                                                                                                                                                                                                                                     |
| <Rn>  | Is the general-purpose base register, encoded in the "Rn" field. For PC use see <a href="#">C2.4.74 LDRSH (literal)</a> on page 490.                                                                                                                                                                                                                                                                                                              |
| +/-   | Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in the "U" field. It can have the following values: <ul style="list-style-type: none"><li>- when U = 0</li><li>+ when U = 1</li></ul>                                                                                                                                                                                               |
| +     | Specifies the offset is added to the base register.                                                                                                                                                                                                                                                                                                                                                                                               |
| <imm> | For the post-indexed or pre-indexed variant: is an 8-bit unsigned immediate byte offset, in the range 0 to 255, encoded in the "imm8" field.<br>For encoding T1: is an optional 12-bit unsigned immediate byte offset, in the range 0 to 4095, defaulting to 0 and encoded in the "imm12" field.<br>For encoding T2: is an optional 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 and encoded in the "imm8" field. |

## Operation for all encodings

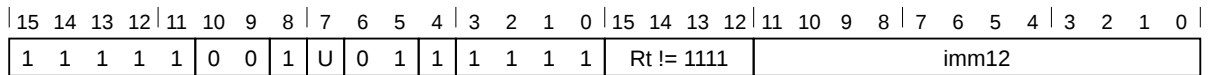
```
1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 offset_addr = if add then (R[n] + imm32) else (R[n] - imm32);
4 address = if index then offset_addr else R[n];
5 // Determine if the stack pointer limit should be checked
6 if n == 13 && wback then
7 (limit, applylimit) = LookUpSPLim(LookUpSP());
8 else
9 applylimit = FALSE;
10 // Memory operation only performed if limit not violated
11 if !applylimit || (UInt(offset_addr) >= UInt(limit)) then
12 R[t] = SignExtend(MemU[address,2], 32);
13
14 // If the stack pointer is being updated a fault will be raised if
15 // the limit is violated
16 if wback then RSPCheck[n] = offset_addr;
```

### C2.4.74 LDRSH (literal)

Load Register Signed Halfword (literal). Load Register Signed Halfword (literal) calculates an address from the PC value and an immediate offset, loads a halfword from memory, sign-extends it to form a 32-bit word, and writes it to a register.

#### T1

Armv8-M Main Extension only



#### T1 variant

```
LDRSH{<c>}{<q>} <Rt>, <label>
 // Preferred syntax
LDRSH{<c>}{<q>} <Rt>, [PC, #{+/-}<imm>]
 // Alternative syntax
```

#### Decode for this encoding

```
1 if Rt == '1111' then SEE "Related encodings";
2 if !HaveMainExt() then UNDEFINED;
3 t = UInt(Rt); imm32 = ZeroExtend(imm12, 32); add = (U == '1');
4 if t == 13 then UNPREDICTABLE;
```

#### Assembler symbols for all encodings

<c> See [C1.2.5 Standard assembler syntax fields](#) on page 309.

<q> See [C1.2.5 Standard assembler syntax fields](#) on page 309.

<Rt> Is the general-purpose register to be transferred, encoded in the "Rt" field.

<label> The label of the literal data item that is to be loaded into <Rt>. The assembler calculates the required value of the offset from the [Align](#)(PC, 4) value of the instruction to this label. Permitted values of the offset are -4095 to 4095. If the offset is zero or positive, imm32 is equal to the offset and add == TRUE, encoded as U == 1. If the offset is negative, imm32 is equal to minus the offset and add == FALSE, encoded as U == 0.

+/- Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in the "U" field. It can have the following values:

- when U = 0
- + when U = 1

<imm> Is a 12-bit unsigned immediate byte offset, in the range 0 to 4095, encoded in the "imm12" field.

#### Operation for all encodings

```
1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 base = Align(PC, 4);
4 address = if add then (base + imm32) else (base - imm32);
5 data = MemU[address, 2];
6 R[t] = SignExtend(data, 32);
```

### C2.4.75 LDRSH (register)

Load Register Signed Halfword (register). Load Register Signed Halfword (register) calculates an address from a base register value and an offset register value, loads a halfword from memory, sign-extends it to form a 32-bit word, and writes it to a register. The offset register value can be shifted left by 0, 1, 2, or 3 bits.

#### T1

Armv8-M

|    |    |    |    |    |    |   |   |    |   |    |   |   |   |   |    |
|----|----|----|----|----|----|---|---|----|---|----|---|---|---|---|----|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7  | 6 | 5  | 4 | 3 | 2 | 1 | 0  |
| 0  | 1  | 0  | 1  | 1  | 1  | 1 |   | Rm |   | Rn |   |   |   |   | Rt |

#### T1 variant

LDRSH{<c>}{<q>} <Rt>, [<Rn>, {+}<Rm>]

#### Decode for this encoding

```
1 t = UInt(Rt); n = UInt(Rn); m = UInt(Rm);
2 index = TRUE; add = TRUE; wback = FALSE;
3 (shift_t, shift_n) = (SRTYPE_LSL, 0);
```

#### T2

Armv8-M Main Extension only

|    |    |    |    |    |    |   |   |   |   |   |   |            |            |   |   |    |    |    |    |    |      |   |   |   |   |    |   |   |   |   |   |
|----|----|----|----|----|----|---|---|---|---|---|---|------------|------------|---|---|----|----|----|----|----|------|---|---|---|---|----|---|---|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3          | 2          | 1 | 0 | 15 | 14 | 13 | 12 | 11 | 10   | 9 | 8 | 7 | 6 | 5  | 4 | 3 | 2 | 1 | 0 |
| 1  | 1  | 1  | 1  | 1  | 0  | 0 | 1 | 0 | 0 | 1 | 1 | Rn != 1111 | Rt != 1111 | 0 | 0 | 0  | 0  | 0  | 0  | 0  | imm2 |   |   |   |   | Rm |   |   |   |   |   |

#### T2 variant

LDRSH{<c>}.W <Rt>, [<Rn>, {+}<Rm>]  
 // <Rt>, <Rn>, <Rm> can be represented in T1  
 LDRSH{<c>}{<q>} <Rt>, [<Rn>, {+}<Rm>{, LSL #<imm>}]

#### Decode for this encoding

```
1 if Rn == '1111' then SEE "LDRSH (literal)";
2 if Rt == '1111' then SEE "Related encodings";
3 if !HaveMainExt() then UNDEFINED;
4 t = UInt(Rt); n = UInt(Rn); m = UInt(Rm);
5 index = TRUE; add = TRUE; wback = FALSE;
6 (shift_t, shift_n) = (SRTYPE_LSL, UInt(imm2));
7 if t == 13 || m IN {13,15} then UNPREDICTABLE;
```

#### Assembler symbols for all encodings

|       |                                                                                                                                                                                   |
|-------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <c>   | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                                                                                                          |
| <q>   | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                                                                                                          |
| <Rt>  | Is the general-purpose register to be transferred, encoded in the "Rt" field.                                                                                                     |
| <Rn>  | Is the general-purpose base register, encoded in the "Rn" field.                                                                                                                  |
| +     | Specifies the index register is added to the base register.                                                                                                                       |
| <Rm>  | Is the general-purpose index register, encoded in the "Rm" field.                                                                                                                 |
| <imm> | If present, the size of the left shift to apply to the value from <Rm>, in the range 1-3. <imm> is encoded in imm2. If absent, no shift is specified and imm2 is encoded as 0b00. |

#### Operation for all encodings

```
1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 offset = Shift(R[m], shift_t, shift_n, APSR.C);
4 offset_addr = if add then (R[n] + offset) else (R[n] - offset);
5 address = if index then offset_addr else R[n];
6 data = MemU(address, 2);
7 if wback then R[n] = offset_addr;
8 R[t] = SignExtend(data, 32);
```





## C2.4.77 LDRT

Load Register Unprivileged. Load Register Unprivileged calculates an address from a base register value and an immediate offset, loads a word from memory, and writes it to a register.

When privileged software uses an **LDRT** instruction, the memory access is restricted as if the software was unprivileged.

### T1

*Armv8-M Main Extension only*

|    |    |    |    |    |    |   |   |   |   |   |   |            |   |   |   |    |    |    |    |    |    |   |   |      |   |   |   |   |   |   |   |
|----|----|----|----|----|----|---|---|---|---|---|---|------------|---|---|---|----|----|----|----|----|----|---|---|------|---|---|---|---|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3          | 2 | 1 | 0 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7    | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 1  | 1  | 1  | 1  | 1  | 0  | 0 | 0 | 0 | 1 | 0 | 1 | Rn != 1111 |   |   |   | Rt |    |    |    | 1  | 1  | 1 | 0 | imm8 |   |   |   |   |   |   |   |

### T1 variant

LDRT{<c>}{<q>} <Rt>, [<Rn> {, #<+><imm>}]

### Decode for this encoding

```

1 if Rn == '1111' then SEE "LDR (literal)";
2 if !HaveMainExt() then UNDEFINED;
3 t = UInt(Rt); n = UInt(Rn); postindex = FALSE; add = TRUE;
4 register_form = FALSE; imm32 = ZeroExtend(imm8, 32);
5 if t IN {13,15} then UNPREDICTABLE;

```

### Assembler symbols for all encodings

|       |                                                                                                                              |
|-------|------------------------------------------------------------------------------------------------------------------------------|
| <c>   | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                                                     |
| <q>   | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                                                     |
| <Rt>  | Is the general-purpose register to be transferred, encoded in the "Rt" field.                                                |
| <Rn>  | Is the general-purpose base register, encoded in the "Rn" field.                                                             |
| +     | Specifies the offset is added to the base register.                                                                          |
| <imm> | Is an optional 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 and encoded in the "imm8" field. |

### Operation for all encodings

```

1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 address = R[n] + imm32;
4 data = MemU_unpriv(address,4);
5 R[t] = data;

```

### C2.4.78 LSL (immediate)

Logical Shift Left (immediate). Logical Shift Left (immediate) shifts a register value left by an immediate number of bits, shifting in zeros, and writes the result to the destination register.

This instruction is an alias of the [MOV \(register\)](#) instruction. This means that:

- The encodings in this description are named to match the encodings of [MOV \(register\)](#).
- The description of [MOV \(register\)](#) gives the operational pseudocode for this instruction.

#### T2

*Armv8-M Main Extension only*

|    |    |    |         |               |    |   |   |    |   |   |   |    |   |   |   |
|----|----|----|---------|---------------|----|---|---|----|---|---|---|----|---|---|---|
| 15 | 14 | 13 | 12      | 11            | 10 | 9 | 8 | 7  | 6 | 5 | 4 | 3  | 2 | 1 | 0 |
| 0  | 0  | 0  | op = 00 | imm5 != 00000 |    |   |   | Rm |   |   |   | Rd |   |   |   |

#### T2 variant

```
LSL<c>{<q>} {<Rd>}, <Rm>, #<imm>
// Inside IT block
```

**is equivalent to**

```
MOV<c>{<q>} <Rd>, <Rm>, LSL #<imm>
```

**and is the preferred disassembly when `InITBlock()`.**

#### T3

*Armv8-M Main Extension only*

|       |    |    |    |    |    |   |   |   |   |   |   |              |   |   |   |     |      |    |    |    |    |   |   |   |      |   |    |   |   |   |   |
|-------|----|----|----|----|----|---|---|---|---|---|---|--------------|---|---|---|-----|------|----|----|----|----|---|---|---|------|---|----|---|---|---|---|
| 15    | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3            | 2 | 1 | 0 | 15  | 14   | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6    | 5 | 4  | 3 | 2 | 1 | 0 |
| 1     | 1  | 1  | 0  | 1  | 0  | 1 | 0 | 0 | 0 | 1 | 0 | 1            | 1 | 1 | 1 | (0) | imm3 |    |    |    | Rd |   |   |   | imm2 |   | Rm |   |   |   |   |
| S = 0 |    |    |    |    |    |   |   |   |   |   |   | sr_type = 00 |   |   |   |     |      |    |    |    |    |   |   |   |      |   |    |   |   |   |   |

#### MOV, shift or rotate by value variant

```
LSL<c>.W {<Rd>}, <Rm>, #<imm>
// Inside IT block, and <Rd>, <Rm>, <imm> can be represented in T2
```

**is equivalent to**

```
MOV{<c>}{<q>} <Rd>, <Rm>, LSL #<imm>
```

**and is always the preferred disassembly.**

#### MOV, shift or rotate by value variant

```
LSL{<c>}{<q>} {<Rd>}, <Rm>, #<imm>
```

**is equivalent to**

```
MOV{<c>}{<q>} <Rd>, <Rm>, LSL #<imm>
```

**and is always the preferred disassembly.**

#### Assembler symbols for all encodings

<c> See [C1.2.5 Standard assembler syntax fields](#) on page 309.

<q> See [C1.2.5 Standard assembler syntax fields](#) on page 309.

|       |                                                                                                                                                                                           |
|-------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <Rd>  | Is the general-purpose destination register, encoded in the "Rd" field.                                                                                                                   |
| <Rm>  | Is the general-purpose source register, encoded in the "Rm" field.                                                                                                                        |
| <imm> | For encoding T2: is the shift amount, in the range 1 to 31, encoded in the "imm5" field.<br>For encoding T3: is the shift amount, in the range 1 to 31, encoded in the "imm3:imm2" field. |

### Operation for all encodings

The description of [MOV \(register\)](#) gives the operational pseudocode for this instruction.

### C2.4.79 LSL (register)

Logical Shift Left (register). Logical Shift Left (register) shifts a register value left by a variable number of bits, shifting in zeros, and writes the result to the destination register. The variable number of bits is read from the bottom byte of a register.

This instruction is an alias of the [MOV, MOVS \(register-shifted register\)](#) instruction. This means that:

- The encodings in this description are named to match the encodings of [MOV, MOVS \(register-shifted register\)](#).
- The description of [MOV, MOVS \(register-shifted register\)](#) gives the operational pseudocode for this instruction.

#### T1

*Armv8-M Main Extension only*

|    |    |    |    |    |    |           |   |    |   |   |   |     |   |   |   |
|----|----|----|----|----|----|-----------|---|----|---|---|---|-----|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9         | 8 | 7  | 6 | 5 | 4 | 3   | 2 | 1 | 0 |
| 0  | 1  | 0  | 0  | 0  | 0  | op = 0010 |   | Rs |   |   |   | Rdm |   |   |   |

#### Logical shift left variant

```
LSL<c>{<q>} {<Rdm>}, <Rdm>, <Rs>
// Inside IT block
```

#### is equivalent to

```
MOV<c>{<q>} <Rdm>, <Rdm>, LSL <Rs>
```

**and is the preferred disassembly when `InITBlock()`.**

#### T2

*Armv8-M Main Extension only*

|    |    |    |    |    |    |   |   |   |                       |   |   |   |    |   |   |    |    |    |    |    |    |   |   |   |   |   |   |   |    |   |   |  |
|----|----|----|----|----|----|---|---|---|-----------------------|---|---|---|----|---|---|----|----|----|----|----|----|---|---|---|---|---|---|---|----|---|---|--|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6                     | 5 | 4 | 3 | 2  | 1 | 0 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2  | 1 | 0 |  |
| 1  | 1  | 1  | 1  | 1  | 0  | 1 | 0 | 0 | sr_type = 00<br>S = 0 |   |   |   | Rm |   |   |    | 1  | 1  | 1  | 1  | Rd |   |   |   | 0 | 0 | 0 | 0 | Rs |   |   |  |

#### Non flag setting variant

```
LSL<c>.W {<Rd>}, <Rm>, <Rs>
// Inside IT block, and <Rd>, <Rm>, <sr_type>, <Rs> can be represented in T1
```

#### is equivalent to

```
MOV{<c>}{<q>} <Rd>, <Rm>, LSL <Rs>
```

**and is always the preferred disassembly.**

#### Non flag setting variant

```
LSL{<c>}{<q>} {<Rd>}, <Rm>, <Rs>
```

#### is equivalent to

```
MOV{<c>}{<q>} <Rd>, <Rm>, LSL <Rs>
```

**and is always the preferred disassembly.**

### Assembler symbols for all encodings

|       |                                                                                                                       |
|-------|-----------------------------------------------------------------------------------------------------------------------|
| <c>   | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                                              |
| <q>   | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                                              |
| <Rdm> | Is the first general-purpose source register and the destination register, encoded in the "Rdm" field.                |
| <Rd>  | Is the general-purpose destination register, encoded in the "Rd" field.                                               |
| <Rm>  | Is the first general-purpose source register, encoded in the "Rm" field.                                              |
| <Rs>  | Is the second general-purpose source register holding a shift amount in its bottom 8 bits, encoded in the "Rs" field. |

### Operation for all encodings

The description of [MOV, MOVS \(register-shifted register\)](#) gives the operational pseudocode for this instruction.

### C2.4.80 LSLS (immediate)

Logical Shift Left, Setting flags (immediate). Logical Shift Left, Setting flags (immediate) shifts a register value left by an immediate number of bits, shifting in zeros, writes the result to the destination register, and updates the condition flags based on the result.

This instruction is an alias of the [MOV \(register\)](#) instruction. This means that:

- The encodings in this description are named to match the encodings of [MOV \(register\)](#).
- The description of [MOV \(register\)](#) gives the operational pseudocode for this instruction.

#### T2

Armv8-M

|    |    |    |         |               |    |   |   |    |   |   |   |    |   |   |   |
|----|----|----|---------|---------------|----|---|---|----|---|---|---|----|---|---|---|
| 15 | 14 | 13 | 12      | 11            | 10 | 9 | 8 | 7  | 6 | 5 | 4 | 3  | 2 | 1 | 0 |
| 0  | 0  | 0  | op = 00 | imm5 != 00000 |    |   |   | Rm |   |   |   | Rd |   |   |   |

#### T2 variant

```
LSLS{<q>} {<Rd>}, <Rm>, #<imm>
// Outside IT block
```

is equivalent to

```
MOVS{<q>} <Rd>, <Rm>, LSL #<imm>
```

and is the preferred disassembly when `!InITBlock()`.

#### T3

Armv8-M Main Extension only

|       |    |    |    |    |    |   |   |   |   |   |   |              |   |   |   |     |      |    |    |    |    |   |   |   |      |   |   |   |    |   |   |  |
|-------|----|----|----|----|----|---|---|---|---|---|---|--------------|---|---|---|-----|------|----|----|----|----|---|---|---|------|---|---|---|----|---|---|--|
| 15    | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3            | 2 | 1 | 0 | 15  | 14   | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6    | 5 | 4 | 3 | 2  | 1 | 0 |  |
| 1     | 1  | 1  | 0  | 1  | 0  | 1 | 0 | 0 | 0 | 1 | 0 | 1            | 1 | 1 | 1 | (0) | imm3 |    |    |    | Rd |   |   |   | imm2 |   |   |   | Rm |   |   |  |
| S = 1 |    |    |    |    |    |   |   |   |   |   |   | sr_type = 00 |   |   |   |     |      |    |    |    |    |   |   |   |      |   |   |   |    |   |   |  |

#### MOVS, shift or rotate by value variant

```
LSLS.W {<Rd>}, <Rm>, #<imm>
// Outside IT block, and <Rd>, <Rm>, <imm> can be represented in T2
```

is equivalent to

```
MOVS{<c>}{<q>} <Rd>, <Rm>, LSL #<imm>
```

and is always the preferred disassembly.

#### MOVS, shift or rotate by value variant

```
LSLS{<c>}{<q>} {<Rd>}, <Rm>, #<imm>
```

is equivalent to

```
MOVS{<c>}{<q>} <Rd>, <Rm>, LSL #<imm>
```

and is always the preferred disassembly.

#### Assembler symbols for all encodings

<c> See [C1.2.5 Standard assembler syntax fields](#) on page 309.

|       |                                                                                                                                                                                           |
|-------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <q>   | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                                                                                                                  |
| <Rd>  | Is the general-purpose destination register, encoded in the "Rd" field.                                                                                                                   |
| <Rm>  | Is the general-purpose source register, encoded in the "Rm" field.                                                                                                                        |
| <imm> | For encoding T2: is the shift amount, in the range 1 to 31, encoded in the "imm5" field.<br>For encoding T3: is the shift amount, in the range 1 to 31, encoded in the "imm3:imm2" field. |

### Operation for all encodings

The description of [MOV \(register\)](#) gives the operational pseudocode for this instruction.



### C2.4.81 LSLS (register)

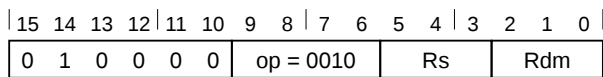
Logical Shift Left, Setting flags (register). Logical Shift Left, Setting flags (register) shifts a register value left by a variable number of bits, shifting in zeros, writes the result to the destination register, and updates the condition flags based on the result. The variable number of bits is read from the bottom byte of a register.

This instruction is an alias of the [MOV, MOVS \(register-shifted register\)](#) instruction. This means that:

- The encodings in this description are named to match the encodings of [MOV, MOVS \(register-shifted register\)](#).
- The description of [MOV, MOVS \(register-shifted register\)](#) gives the operational pseudocode for this instruction.

#### T1

*Armv8-M*



#### Logical shift left variant

```
LSLS{<q>} {<Rdm>, } <Rdm>, <Rs>
// Outside IT block
```

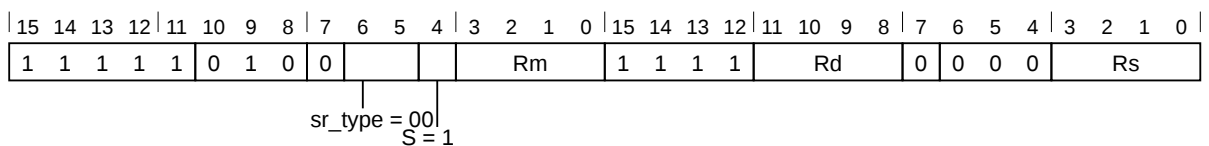
#### is equivalent to

```
MOVS{<q>} <Rdm>, <Rdm>, LSL <Rs>
```

**and is the preferred disassembly when !InITBlock().**

#### T2

*Armv8-M Main Extension only*



#### Flag setting variant

```
LSLS.W {<Rd>, } <Rm>, <Rs>
// Outside IT block, and <Rd>, <Rm>, <sr_type>, <Rs> can be represented in T1
```

#### is equivalent to

```
MOVS{<c>}{<q>} <Rd>, <Rm>, LSL <Rs>
```

**and is always the preferred disassembly.**

#### Flag setting variant

```
LSLS{<c>}{<q>} {<Rd>, } <Rm>, <Rs>
```

#### is equivalent to

```
MOVS{<c>}{<q>} <Rd>, <Rm>, LSL <Rs>
```

**and is always the preferred disassembly.**

### Assembler symbols for all encodings

|       |                                                                                                                       |
|-------|-----------------------------------------------------------------------------------------------------------------------|
| <c>   | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                                              |
| <q>   | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                                              |
| <Rdm> | Is the first general-purpose source register and the destination register, encoded in the "Rdm" field.                |
| <Rd>  | Is the general-purpose destination register, encoded in the "Rd" field.                                               |
| <Rm>  | Is the first general-purpose source register, encoded in the "Rm" field.                                              |
| <Rs>  | Is the second general-purpose source register holding a shift amount in its bottom 8 bits, encoded in the "Rs" field. |

### Operation for all encodings

The description of [MOV, MOVS \(register-shifted register\)](#) gives the operational pseudocode for this instruction.

### C2.4.82 LSR (immediate)

Logical Shift Right (immediate). Logical Shift Right (immediate) shifts a register value right by an immediate number of bits, shifting in zeros, and writes the result to the destination register.

This instruction is an alias of the [MOV \(register\)](#) instruction. This means that:

- The encodings in this description are named to match the encodings of [MOV \(register\)](#).
- The description of [MOV \(register\)](#) gives the operational pseudocode for this instruction.

#### T2

*Armv8-M Main Extension only*

|    |    |    |         |      |    |   |   |    |   |   |    |   |   |   |   |
|----|----|----|---------|------|----|---|---|----|---|---|----|---|---|---|---|
| 15 | 14 | 13 | 12      | 11   | 10 | 9 | 8 | 7  | 6 | 5 | 4  | 3 | 2 | 1 | 0 |
| 0  | 0  | 0  | op = 01 | imm5 |    |   |   | Rm |   |   | Rd |   |   |   |   |

#### T2 variant

```
LSR<c>{<q>} {<Rd>}, <Rm>, #<imm>
// Inside IT block
```

#### is equivalent to

```
MOV<c>{<q>} <Rd>, <Rm>, LSR #<imm>
```

**and is the preferred disassembly when `InITBlock()`.**

#### T3

*Armv8-M Main Extension only*

|       |    |    |    |    |    |   |   |   |   |   |   |              |   |   |   |     |      |    |    |    |    |   |      |   |    |   |   |   |   |   |   |
|-------|----|----|----|----|----|---|---|---|---|---|---|--------------|---|---|---|-----|------|----|----|----|----|---|------|---|----|---|---|---|---|---|---|
| 15    | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3            | 2 | 1 | 0 | 15  | 14   | 13 | 12 | 11 | 10 | 9 | 8    | 7 | 6  | 5 | 4 | 3 | 2 | 1 | 0 |
| 1     | 1  | 1  | 0  | 1  | 0  | 1 | 0 | 0 | 0 | 1 | 0 | 1            | 1 | 1 | 1 | (0) | imm3 |    |    | Rd |    |   | imm2 |   | Rm |   |   |   |   |   |   |
| S = 0 |    |    |    |    |    |   |   |   |   |   |   | sr_type = 01 |   |   |   |     |      |    |    |    |    |   |      |   |    |   |   |   |   |   |   |

#### MOV, shift or rotate by value variant

```
LSR<c>.W {<Rd>}, <Rm>, #<imm>
// Inside IT block, and <Rd>, <Rm>, <imm> can be represented in T2
```

#### is equivalent to

```
MOV{<c>}{<q>} <Rd>, <Rm>, LSR #<imm>
```

**and is always the preferred disassembly.**

#### MOV, shift or rotate by value variant

```
LSR{<c>}{<q>} {<Rd>}, <Rm>, #<imm>
```

#### is equivalent to

```
MOV{<c>}{<q>} <Rd>, <Rm>, LSR #<imm>
```

**and is always the preferred disassembly.**

#### Assembler symbols for all encodings

<c> See [C1.2.5 Standard assembler syntax fields](#) on page 309.

<q> See [C1.2.5 Standard assembler syntax fields](#) on page 309.

|       |                                                                                                                                                                                                                                 |
|-------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <Rd>  | Is the general-purpose destination register, encoded in the "Rd" field.                                                                                                                                                         |
| <Rm>  | Is the general-purpose source register, encoded in the "Rm" field.                                                                                                                                                              |
| <imm> | For encoding T2: is the shift amount, in the range 1 to 32, encoded in the "imm5" field as <imm> modulo 32.<br>For encoding T3: is the shift amount, in the range 1 to 32, encoded in the "imm3:imm2" field as <imm> modulo 32. |

### Operation for all encodings

The description of [MOV \(register\)](#) gives the operational pseudocode for this instruction.

### C2.4.83 LSR (register)

Logical Shift Right (register). Logical Shift Right (register) shifts a register value right by a variable number of bits, shifting in zeros, and writes the result to the destination register. The variable number of bits is read from the bottom byte of a register.

This instruction is an alias of the [MOV, MOVS \(register-shifted register\)](#) instruction. This means that:

- The encodings in this description are named to match the encodings of [MOV, MOVS \(register-shifted register\)](#).
- The description of [MOV, MOVS \(register-shifted register\)](#) gives the operational pseudocode for this instruction.

#### T1

*Armv8-M Main Extension only*

|    |    |    |    |    |    |           |   |   |   |    |   |   |     |   |   |
|----|----|----|----|----|----|-----------|---|---|---|----|---|---|-----|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9         | 8 | 7 | 6 | 5  | 4 | 3 | 2   | 1 | 0 |
| 0  | 1  | 0  | 0  | 0  | 0  | op = 0011 |   |   |   | Rs |   |   | Rdm |   |   |

#### Logical shift right variant

```
LSR<c>{<q>} {<Rdm>}, <Rdm>, <Rs>
// Inside IT block
```

#### is equivalent to

```
MOV<c>{<q>} <Rdm>, <Rdm>, LSR <Rs>
```

**and is the preferred disassembly when `InITBlock()`.**

#### T2

*Armv8-M Main Extension only*

|    |    |    |    |    |    |   |   |   |   |   |   |    |   |   |   |    |    |    |    |    |    |   |   |   |   |    |   |   |   |   |   |
|----|----|----|----|----|----|---|---|---|---|---|---|----|---|---|---|----|----|----|----|----|----|---|---|---|---|----|---|---|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3  | 2 | 1 | 0 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5  | 4 | 3 | 2 | 1 | 0 |
| 1  | 1  | 1  | 1  | 1  | 0  | 1 | 0 | 0 |   |   |   | Rm |   |   | 1 | 1  | 1  | 1  | Rd |    |    | 0 | 0 | 0 | 0 | Rs |   |   |   |   |   |

sr\_type  
 S = 0

#### Non flag setting variant

```
LSR<c>.W {<Rd>}, <Rm>, <Rs>
// Inside IT block, and <Rd>, <Rm>, <type>, <Rs> can be represented in T1
```

#### is equivalent to

```
MOV{<c>}{<q>} <Rd>, <Rm>, LSR <Rs>
```

**and is always the preferred disassembly.**

#### Non flag setting variant

```
LSR{<c>}{<q>} {<Rd>}, <Rm>, <Rs>
```

#### is equivalent to

```
MOV{<c>}{<q>} <Rd>, <Rm>, LSR <Rs>
```

**and is always the preferred disassembly.**

### Assembler symbols for all encodings

|       |                                                                                                                       |
|-------|-----------------------------------------------------------------------------------------------------------------------|
| <c>   | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                                              |
| <q>   | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                                              |
| <Rdm> | Is the first general-purpose source register and the destination register, encoded in the "Rdm" field.                |
| <Rd>  | Is the general-purpose destination register, encoded in the "Rd" field.                                               |
| <Rm>  | Is the first general-purpose source register, encoded in the "Rm" field.                                              |
| <Rs>  | Is the second general-purpose source register holding a shift amount in its bottom 8 bits, encoded in the "Rs" field. |

### Operation for all encodings

The description of [MOV, MOVS \(register-shifted register\)](#) gives the operational pseudocode for this instruction.

### C2.4.84 LSRS (immediate)

Logical Shift Right, Setting flags (immediate). Logical Shift Right, Setting flags (immediate) shifts a register value right by an immediate number of bits, shifting in zeros, writes the result to the destination register, and updates the condition flags based on the result.

This instruction is an alias of the [MOV \(register\)](#) instruction. This means that:

- The encodings in this description are named to match the encodings of [MOV \(register\)](#).
- The description of [MOV \(register\)](#) gives the operational pseudocode for this instruction.

#### T2

Armv8-M

|    |    |    |         |      |    |   |   |    |   |   |    |   |   |   |   |
|----|----|----|---------|------|----|---|---|----|---|---|----|---|---|---|---|
| 15 | 14 | 13 | 12      | 11   | 10 | 9 | 8 | 7  | 6 | 5 | 4  | 3 | 2 | 1 | 0 |
| 0  | 0  | 0  | op = 01 | imm5 |    |   |   | Rm |   |   | Rd |   |   |   |   |

#### T2 variant

```
LSRS{<q>} {<Rd>}, <Rm>, #<imm>
// Outside IT block
```

is equivalent to

```
MOVS{<q>} <Rd>, <Rm>, LSR #<imm>
```

and is the preferred disassembly when `!InITBlock()`.

#### T3

Armv8-M Main Extension only

|       |    |    |    |    |    |   |   |   |   |   |   |              |   |   |   |     |      |    |    |    |    |   |      |   |    |   |   |   |   |   |   |
|-------|----|----|----|----|----|---|---|---|---|---|---|--------------|---|---|---|-----|------|----|----|----|----|---|------|---|----|---|---|---|---|---|---|
| 15    | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3            | 2 | 1 | 0 | 15  | 14   | 13 | 12 | 11 | 10 | 9 | 8    | 7 | 6  | 5 | 4 | 3 | 2 | 1 | 0 |
| 1     | 1  | 1  | 0  | 1  | 0  | 1 | 0 | 0 | 0 | 1 | 0 | 1            | 1 | 1 | 1 | (0) | imm3 |    |    | Rd |    |   | imm2 |   | Rm |   |   |   |   |   |   |
| S = 1 |    |    |    |    |    |   |   |   |   |   |   | sr_type = 01 |   |   |   |     |      |    |    |    |    |   |      |   |    |   |   |   |   |   |   |

#### MOVS, shift or rotate by value variant

```
LSRS.W {<Rd>}, <Rm>, #<imm>
// Outside IT block, and <Rd>, <Rm>, <imm> can be represented in T2
```

is equivalent to

```
MOVS{<c>}{<q>} <Rd>, <Rm>, LSR #<imm>
```

and is always the preferred disassembly.

#### MOVS, shift or rotate by value variant

```
LSRS{<c>}{<q>} {<Rd>}, <Rm>, #<imm>
```

is equivalent to

```
MOVS{<c>}{<q>} <Rd>, <Rm>, LSR #<imm>
```

and is always the preferred disassembly.

#### Assembler symbols for all encodings

<c> See [C1.2.5 Standard assembler syntax fields](#) on page 309.

|       |                                                                                                                                                                                                                                 |
|-------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <q>   | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                                                                                                                                                        |
| <Rd>  | Is the general-purpose destination register, encoded in the "Rd" field.                                                                                                                                                         |
| <Rm>  | Is the general-purpose source register, encoded in the "Rm" field.                                                                                                                                                              |
| <imm> | For encoding T2: is the shift amount, in the range 1 to 32, encoded in the "imm5" field as <imm> modulo 32.<br>For encoding T3: is the shift amount, in the range 1 to 32, encoded in the "imm3:imm2" field as <imm> modulo 32. |

### Operation for all encodings

The description of [MOV \(register\)](#) gives the operational pseudocode for this instruction.



### C2.4.85 LSRS (register)

Logical Shift Right, Setting flags (register). Logical Shift Right, Setting flags (register) shifts a register value right by a variable number of bits, shifting in zeros, writes the result to the destination register, and updates the condition flags based on the result. The variable number of bits is read from the bottom byte of a register.

This instruction is an alias of the [MOV, MOVS \(register-shifted register\)](#) instruction. This means that:

- The encodings in this description are named to match the encodings of [MOV, MOVS \(register-shifted register\)](#).
- The description of [MOV, MOVS \(register-shifted register\)](#) gives the operational pseudocode for this instruction.

#### T1

*Armv8-M*

|    |    |    |    |    |    |           |   |   |   |    |   |   |     |   |   |
|----|----|----|----|----|----|-----------|---|---|---|----|---|---|-----|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9         | 8 | 7 | 6 | 5  | 4 | 3 | 2   | 1 | 0 |
| 0  | 1  | 0  | 0  | 0  | 0  | op = 0011 |   |   |   | Rs |   |   | Rdm |   |   |

#### Logical shift right variant

```
LSRS{<q>} {<Rdm>, } <Rdm>, <Rs>
// Outside IT block
```

**is equivalent to**

```
MOVS{<q>} <Rdm>, <Rdm>, LSR <Rs>
```

**and is the preferred disassembly when !InITBlock().**

#### T2

*Armv8-M Main Extension only*

|    |    |    |    |    |    |   |   |   |   |   |   |    |   |   |   |    |    |    |    |    |    |   |   |   |   |    |   |   |   |   |   |
|----|----|----|----|----|----|---|---|---|---|---|---|----|---|---|---|----|----|----|----|----|----|---|---|---|---|----|---|---|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3  | 2 | 1 | 0 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5  | 4 | 3 | 2 | 1 | 0 |
| 1  | 1  | 1  | 1  | 1  | 0  | 1 | 0 | 0 |   |   |   | Rm |   |   | 1 | 1  | 1  | 1  | Rd |    |    | 0 | 0 | 0 | 0 | Rs |   |   |   |   |   |

$\begin{matrix} | \\ \text{sr\_type} = 01 \\ \text{S} = 1 \end{matrix}$

#### Flag setting variant

```
LSRS.W {<Rd>, } <Rm>, <Rs>
// Outside IT block, and <Rd>, <Rm>, <sr_type>, <Rs> can be represented in T1
```

**is equivalent to**

```
MOVS{<c>}{<q>} <Rd>, <Rm>, LSR <Rs>
```

**and is always the preferred disassembly.**

#### Flag setting variant

```
LSRS{<c>}{<q>} {<Rd>, } <Rm>, <Rs>
```

**is equivalent to**

```
MOVS{<c>}{<q>} <Rd>, <Rm>, LSR <Rs>
```

**and is always the preferred disassembly.**

### Assembler symbols for all encodings

|       |                                                                                                                       |
|-------|-----------------------------------------------------------------------------------------------------------------------|
| <c>   | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                                              |
| <q>   | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                                              |
| <Rdm> | Is the first general-purpose source register and the destination register, encoded in the "Rdm" field.                |
| <Rd>  | Is the general-purpose destination register, encoded in the "Rd" field.                                               |
| <Rm>  | Is the first general-purpose source register, encoded in the "Rm" field.                                              |
| <Rs>  | Is the second general-purpose source register holding a shift amount in its bottom 8 bits, encoded in the "Rs" field. |

### Operation for all encodings

The description of [MOV, MOVS \(register-shifted register\)](#) gives the operational pseudocode for this instruction.

## C2.4.86 MCR, MCR2

Move to Coprocessor from Register. Move to Coprocessor from Register passes the value of a general-purpose register to a coprocessor.

If no coprocessor can execute the instruction, a UsageFault exception is generated.

Arm reserves coprocessors CP8 to CP15, and this manual defines the valid instructions when coproc is in this range.

### T1

*Armv8-M Main Extension only*

|    |    |    |    |    |    |   |   |      |   |   |   |   |     |   |   |    |    |    |    |    |        |   |   |   |      |   |   |   |   |     |   |  |  |
|----|----|----|----|----|----|---|---|------|---|---|---|---|-----|---|---|----|----|----|----|----|--------|---|---|---|------|---|---|---|---|-----|---|--|--|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7    | 6 | 5 | 4 | 3 | 2   | 1 | 0 | 15 | 14 | 13 | 12 | 11 | 10     | 9 | 8 | 7 | 6    | 5 | 4 | 3 | 2 | 1   | 0 |  |  |
| 1  | 1  | 1  | 0  | 1  | 1  | 1 | 0 | opc1 |   |   |   | 0 | CRn |   |   |    | Rt |    |    |    | coproc |   |   |   | opc2 |   |   |   | 1 | CRm |   |  |  |

#### T1 variant

MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

#### Decode for this encoding

```
1 if coproc IN '101x' then SEE "Floating-point";
2 if !HaveMainExt() then UNDEFINED;
3 t = UInt(Rt); cp = UInt(coproc);
4 if t == 15 || t == 13 then UNPREDICTABLE;
```

### T2

*Armv8-M Main Extension only*

|    |    |    |    |    |    |   |   |      |   |   |   |   |     |   |   |    |    |    |    |    |        |   |   |   |      |   |   |   |   |     |   |  |  |
|----|----|----|----|----|----|---|---|------|---|---|---|---|-----|---|---|----|----|----|----|----|--------|---|---|---|------|---|---|---|---|-----|---|--|--|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7    | 6 | 5 | 4 | 3 | 2   | 1 | 0 | 15 | 14 | 13 | 12 | 11 | 10     | 9 | 8 | 7 | 6    | 5 | 4 | 3 | 2 | 1   | 0 |  |  |
| 1  | 1  | 1  | 1  | 1  | 1  | 1 | 0 | opc1 |   |   |   | 0 | CRn |   |   |    | Rt |    |    |    | coproc |   |   |   | opc2 |   |   |   | 1 | CRm |   |  |  |

#### T2 variant

MCR2{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

#### Decode for this encoding

```
1 if coproc IN '101x' then UNDEFINED;
2 if !HaveMainExt() then UNDEFINED;
3 t = UInt(Rt); cp = UInt(coproc);
4 if t == 15 || t == 13 then UNPREDICTABLE;
```

#### Assembler symbols for all encodings

|          |                                                                                                             |
|----------|-------------------------------------------------------------------------------------------------------------|
| <c>      | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                                    |
| <q>      | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                                    |
| <coproc> | Is the name of the coprocessor, encoded in the "coproc" field. The generic coprocessor names are p0 to p15. |
| <opc1>   | Is a coprocessor-specific opcode in the range 0 to 7, encoded in the "opc1" field.                          |
| <Rt>     | Is the general-purpose register to be transferred, encoded in the "Rt" field.                               |
| <CRn>    | Is the first coprocessor register, encoded in the "CRn" field.                                              |
| <CRm>    | Is the second coprocessor register, encoded in the "CRm" field.                                             |
| <opc2>   | Is a coprocessor-specific opcode in the range 0 to 7, defaulting to 0 and encoded in the "opc2" field.      |

### Operation for all encodings

```
1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 ExecuteCPCheck(cp);
4 if !Coproced_Accepted(cp, ThisInstr()) then
5 GenerateCoproced_Exception();
6 else
7 Coproc_SendOneWord(R[t], cp, ThisInstr());
```

## C2.4.87 MCRR, MCRR2

Move to Coprocessor from two Registers. Move to Coprocessor from two Registers passes the values of two general-purpose registers to a coprocessor.

If no coprocessor can execute the instruction, a UsageFault exception is generated.

Arm reserves coprocessors CP8 to CP15, and this manual defines the valid instructions when coproc is in this range.

### T1

*Armv8-M Main Extension only*

|    |    |    |    |    |    |   |   |   |   |   |   |     |   |   |   |    |    |    |    |        |    |   |   |      |   |   |   |     |   |   |   |
|----|----|----|----|----|----|---|---|---|---|---|---|-----|---|---|---|----|----|----|----|--------|----|---|---|------|---|---|---|-----|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3   | 2 | 1 | 0 | 15 | 14 | 13 | 12 | 11     | 10 | 9 | 8 | 7    | 6 | 5 | 4 | 3   | 2 | 1 | 0 |
| 1  | 1  | 1  | 0  | 1  | 1  | 0 | 0 | 0 | 1 | 0 | 0 | Rt2 |   |   |   | Rt |    |    |    | coproc |    |   |   | opc1 |   |   |   | CRm |   |   |   |

#### T1 variant

MCRR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <Rt2>, <CRm>

#### Decode for this encoding

```

1 if coproc IN '101x' then SEE "Floating-point";
2 if !HaveMainExt() then UNDEFINED;
3 t = UInt(Rt); t2 = UInt(Rt2); cp = UInt(coproc);
4 if t == 15 || t2 == 15 then UNPREDICTABLE;
5 if t == 13 || t2 == 13 then UNPREDICTABLE;

```

### T2

*Armv8-M Main Extension only*

|    |    |    |    |    |    |   |   |   |   |   |   |     |   |   |   |    |    |    |    |        |    |   |   |      |   |   |   |     |   |   |   |
|----|----|----|----|----|----|---|---|---|---|---|---|-----|---|---|---|----|----|----|----|--------|----|---|---|------|---|---|---|-----|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3   | 2 | 1 | 0 | 15 | 14 | 13 | 12 | 11     | 10 | 9 | 8 | 7    | 6 | 5 | 4 | 3   | 2 | 1 | 0 |
| 1  | 1  | 1  | 1  | 1  | 1  | 0 | 0 | 0 | 1 | 0 | 0 | Rt2 |   |   |   | Rt |    |    |    | coproc |    |   |   | opc1 |   |   |   | CRm |   |   |   |

#### T2 variant

MCRR2{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <Rt2>, <CRm>

#### Decode for this encoding

```

1 if coproc IN '101x' then UNDEFINED;
2 if !HaveMainExt() then UNDEFINED;
3 t = UInt(Rt); t2 = UInt(Rt2); cp = UInt(coproc);
4 if t == 15 || t2 == 15 then UNPREDICTABLE;
5 if t == 13 || t2 == 13 then UNPREDICTABLE;

```

#### Assembler symbols for all encodings

|          |                                                                                                             |
|----------|-------------------------------------------------------------------------------------------------------------|
| <c>      | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                                    |
| <q>      | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                                    |
| <coproc> | Is the name of the coprocessor, encoded in the "coproc" field. The generic coprocessor names are p0 to p15. |
| <opc1>   | Is a coprocessor-specific opcode in the range 0 to 15, encoded in the "opc1" field.                         |
| <Rt>     | Is the first general-purpose register to be transferred, encoded in the "Rt" field.                         |
| <Rt2>    | Is the second general-purpose register to be transferred, encoded in the "Rt2" field.                       |
| <CRm>    | Is a coprocessor register, encoded in the "CRm" field.                                                      |

### Operation for all encodings

```
1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 ExecuteCPCheck(cp);
4 if !Coproced_Accepted(cp, ThisInstr()) then
5 GenerateCoproced_Exception();
6 else
7 Coproc_SendTwoWords(R[t2], R[t], cp, ThisInstr());
```

## C2.4.88 MLA

Multiply Accumulate. Multiply Accumulate multiplies two register values, and adds a third register value. The least significant 32 bits of the result are written to the destination register. These 32 bits do not depend on whether signed or unsigned calculations are performed.

### T1

Armv8-M Main Extension only

|    |    |    |    |    |    |   |   |   |   |   |   |    |   |   |   |            |    |    |    |    |    |   |   |         |   |   |   |    |   |   |   |
|----|----|----|----|----|----|---|---|---|---|---|---|----|---|---|---|------------|----|----|----|----|----|---|---|---------|---|---|---|----|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3  | 2 | 1 | 0 | 15         | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7       | 6 | 5 | 4 | 3  | 2 | 1 | 0 |
| 1  | 1  | 1  | 1  | 1  | 0  | 1 | 1 | 0 | 0 | 0 | 0 | Rn |   |   |   | Ra != 1111 |    |    |    | Rd |    |   |   | 0 0 0 0 |   |   |   | Rm |   |   |   |

### T1 variant

MLA{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>

### Decode for this encoding

```

1 if Ra == '1111' then SEE MUL;
2 if !HaveMainExt() then UNDEFINED;
3 d = UInt(Rd); n = UInt(Rn); m = UInt(Rm); a = UInt(Ra); setflags = FALSE;
4 if d IN {13,15} || n IN {13,15} || m IN {13,15} || a == 13 then UNPREDICTABLE;

```

### Assembler symbols for all encodings

|      |                                                                                                   |
|------|---------------------------------------------------------------------------------------------------|
| <c>  | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                          |
| <q>  | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                          |
| <Rd> | Is the general-purpose destination register, encoded in the "Rd" field.                           |
| <Rn> | Is the first general-purpose source register holding the multiplicand, encoded in the "Rn" field. |
| <Rm> | Is the second general-purpose source register holding the multiplier, encoded in the "Rm" field.  |
| <Ra> | Is the third general-purpose source register holding the addend, encoded in the "Ra" field.       |

### Operation for all encodings

```

1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 operand1 = SInt(R[n]); // operand1 = UInt(R[n]) produces the same final results
4 operand2 = SInt(R[m]); // operand2 = UInt(R[m]) produces the same final results
5 addend = SInt(R[a]); // addend = UInt(R[a]) produces the same final results
6 result = operand1 * operand2 + addend;
7 R[d] = result[31:0];
8 if setflags then
9 APSR.N = result[31];
10 APSR.Z = IsZeroBit(result[31:0]);
11 // APSR.C unchanged
12 // APSR.V unchanged

```

## C2.4.89 MLS

Multiply and Subtract. Multiply and Subtract multiplies two register values, and subtracts the least significant 32 bits of the result from a third register value. These 32 bits do not depend on whether signed or unsigned calculations are performed. The result is written to the destination register.

### T1

Armv8-M Main Extension only

|    |    |    |    |    |    |   |   |   |   |   |   |    |   |   |   |    |    |    |    |    |    |   |   |   |   |   |   |    |   |   |   |
|----|----|----|----|----|----|---|---|---|---|---|---|----|---|---|---|----|----|----|----|----|----|---|---|---|---|---|---|----|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3  | 2 | 1 | 0 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3  | 2 | 1 | 0 |
| 1  | 1  | 1  | 1  | 1  | 0  | 1 | 1 | 0 | 0 | 0 | 0 | Rn |   |   |   | Ra |    |    |    | Rd |    |   |   | 0 | 0 | 0 | 1 | Rm |   |   |   |

### T1 variant

MLS{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>

### Decode for this encoding

```

1 if !HaveMainExt() then UNDEFINED;
2 d = UInt(Rd); n = UInt(Rn); m = UInt(Rm); a = UInt(Ra);
3 if d IN {13,15} || n IN {13,15} || m IN {13,15} || a IN {13,15} then UNPREDICTABLE;

```

### Assembler symbols for all encodings

|      |                                                                                                   |
|------|---------------------------------------------------------------------------------------------------|
| <c>  | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                          |
| <q>  | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                          |
| <Rd> | Is the general-purpose destination register, encoded in the "Rd" field.                           |
| <Rn> | Is the first general-purpose source register holding the multiplicand, encoded in the "Rn" field. |
| <Rm> | Is the second general-purpose source register holding the multiplier, encoded in the "Rm" field.  |
| <Ra> | Is the third general-purpose source register holding the minuend, encoded in the "Ra" field.      |

### Operation for all encodings

```

1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 operand1 = SInt(R[n]); // operand1 = UInt(R[n]) produces the same final results
4 operand2 = SInt(R[m]); // operand2 = UInt(R[m]) produces the same final results
5 addend = SInt(R[a]); // addend = UInt(R[a]) produces the same final results
6 result = addend - operand1 * operand2;
7 R[d] = result[31:0];

```



## C2.4.90 MOV (immediate)

Move (immediate). Move (immediate) writes an immediate value to the destination register. It can optionally update the condition flags based on the value.

### T1

Armv8-M

|    |    |    |    |    |    |   |   |   |      |   |   |   |   |   |   |  |
|----|----|----|----|----|----|---|---|---|------|---|---|---|---|---|---|--|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6    | 5 | 4 | 3 | 2 | 1 | 0 |  |
| 0  | 0  | 1  | 0  | 0  | Rd |   |   |   | imm8 |   |   |   |   |   |   |  |

#### T1 variant

```
MOV<c>{<q>} <Rd>, #<imm8>
// Inside IT block
MOVS{<q>} <Rd>, #<imm8>
// Outside IT block
```

#### Decode for this encoding

```
1 d = UInt(Rd); setflags = !InITBlock(); imm32 = ZeroExtend(imm8, 32); carry = APSR.C;
```

### T2

Armv8-M Main Extension only

|    |    |    |    |    |    |   |   |   |   |   |   |   |   |   |   |    |      |    |    |    |    |      |   |   |   |   |   |   |   |   |   |
|----|----|----|----|----|----|---|---|---|---|---|---|---|---|---|---|----|------|----|----|----|----|------|---|---|---|---|---|---|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | 15 | 14   | 13 | 12 | 11 | 10 | 9    | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 1  | 1  | 1  | 1  | 0  | i  | 0 | 0 | 0 | 1 | 0 | S | 1 | 1 | 1 | 1 | 0  | imm3 | Rd |    |    |    | imm8 |   |   |   |   |   |   |   |   |   |

#### MOV variant

Applies when **S** == 0.

```
MOV<c>.W <Rd>, #<const>
// Inside IT block, and <Rd>, <const> can be represented in T1
MOV{<c>}{<q>} <Rd>, #<const>
```

#### MOVS variant

Applies when **S** == 1.

```
MOVS.W <Rd>, #<const>
// Outside IT block, and <Rd>, <const> can be represented in T1
MOVS{<c>}{<q>} <Rd>, #<const>
```

#### Decode for this encoding

```
1 if !HaveMainExt() then UNDEFINED;
2 d = UInt(Rd); setflags = (S == '1'); (imm32, carry) = T32ExpandImm_C(i:imm3:imm8, APSR.C);
3 if d IN {13,15} then UNPREDICTABLE;
```

### T3

Armv8-M

|    |    |    |    |    |    |   |   |   |   |   |   |      |   |   |   |    |      |    |    |    |    |   |   |   |      |   |   |   |   |   |   |  |
|----|----|----|----|----|----|---|---|---|---|---|---|------|---|---|---|----|------|----|----|----|----|---|---|---|------|---|---|---|---|---|---|--|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3    | 2 | 1 | 0 | 15 | 14   | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6    | 5 | 4 | 3 | 2 | 1 | 0 |  |
| 1  | 1  | 1  | 1  | 0  | i  | 1 | 0 | 0 | 1 | 0 | 0 | imm4 |   |   |   | 0  | imm3 |    |    |    | Rd |   |   |   | imm8 |   |   |   |   |   |   |  |

#### T3 variant

```
MOV{<c>}{<q>} <Rd>, #<imm16>
// <imm16> cannot be represented in T1 or T2
MOVW{<c>}{<q>} <Rd>, #<imm16>
// <imm16> can be represented in T1 or T2
```

### Decode for this encoding

```
1 d = UInt(Rd); setflags = FALSE; imm32 = ZeroExtend(imm4:i:imm3:imm8, 32); carry = bit
 UNKNOWN;
2 if d IN {13,15} then UNPREDICTABLE;
```

### Assembler symbols for all encodings

|         |                                                                                                                                                                                                |
|---------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <c>     | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                                                                                                                       |
| <q>     | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                                                                                                                       |
| <Rd>    | Is the general-purpose destination register, encoded in the "Rd" field.                                                                                                                        |
| <imm8>  | Is an 8-bit unsigned immediate, in the range 0 to 255, encoded in the "imm8" field.                                                                                                            |
| <imm16> | Is a 16-bit unsigned immediate, in the range 0 to 65535, encoded in the "imm4:i:imm3:imm8" field.                                                                                              |
| <const> | Is an immediate value derived from the 12-bit immediate that is encoded in the 'i:imm3:imm8' field. See <a href="#">C1.5 Modified immediate constants</a> on page 324 for the range of values. |

### Operation for all encodings

```
1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 result = imm32;
4 R[d] = result;
5 if setflags then
6 APSR.N = result[31];
7 APSR.Z = IsZeroBit(result);
8 APSR.C = carry;
9 // APSR.V unchanged
```

## C2.4.91 MOV (register)

Move (register). Move (register) copies a value from a register to the destination register. It can optionally update the condition flags based on the value.

This instruction is used by the aliases [ASRS \(immediate\)](#), [ASR \(immediate\)](#), [LSLS \(immediate\)](#), [LSL \(immediate\)](#), [LSRS \(immediate\)](#), [LSR \(immediate\)](#), [RORS \(immediate\)](#), [ROR \(immediate\)](#), [RRXS](#), [RRX](#).

### T1

Armv8-M

|    |    |    |    |    |    |   |   |   |    |   |   |   |    |   |   |
|----|----|----|----|----|----|---|---|---|----|---|---|---|----|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6  | 5 | 4 | 3 | 2  | 1 | 0 |
| 0  | 1  | 0  | 0  | 0  | 1  | 1 | 0 | D | Rm |   |   |   | Rd |   |   |

#### T1 variant

MOV{<c>}{<q>} <Rd>, <Rm>

#### Decode for this encoding

```

1 d = UInt(D:Rd); m = UInt(Rm); setflags = FALSE;
2 (shift_t, shift_n) = (SRTYPE_LSL, 0);
3 if HaveMainExt() then
4 if d == 15 && InITBlock() && !LastInITBlock() then UNPREDICTABLE;
```

### T2

Armv8-M

|    |    |    |          |      |    |   |   |    |   |   |   |    |   |   |   |
|----|----|----|----------|------|----|---|---|----|---|---|---|----|---|---|---|
| 15 | 14 | 13 | 12       | 11   | 10 | 9 | 8 | 7  | 6 | 5 | 4 | 3  | 2 | 1 | 0 |
| 0  | 0  | 0  | op != 11 | imm5 |    |   |   | Rm |   |   |   | Rd |   |   |   |

#### T2 variant

```

MOV<c>{<q>} <Rd>, <Rm> {, <shift> #<amount>}
// Inside IT block
MOVS{<q>} <Rd>, <Rm> {, <shift> #<amount>}
// Outside IT block
```

#### Decode for this encoding

```

1 if op == '11' then SEE "Related encodings";
2 d = UInt(Rd); m = UInt(Rm); setflags = !InITBlock();
3 (shift_t, shift_n) = DecodeImmShift(op, imm5);
4 if op == '00' && imm5 == '00000' && InITBlock() then UNPREDICTABLE;
```

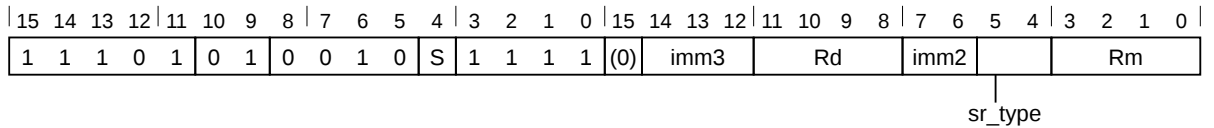
#### CONSTRAINED UNPREDICTABLE behavior

If `op == '00' && imm5 == '00000' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passed its condition code check.
- The instruction executes as [NOP](#), as if it failed its condition code check.
- The instruction executes as `MOV Rd, Rm`.

## T3

Armv8-M Main Extension only



### MOV, rotate right with extend variant

Applies when **S == 0** && **imm3 == 000** && **imm2 == 00** && **sr\_type == 11**.

MOV{<c>}{<q>} <Rd>, <Rm>, RRX

### MOV, shift or rotate by value variant

Applies when **S == 0** && **!(imm3 == 000 && imm2 == 00 && sr\_type == 11)**.

```
MOV<c>.W <Rd>, <Rm> {, <shift> #<amount>}
// Inside IT block
// and <Rd>, <Rm>, <shift>, <amount> can be represented in T1 or T2
MOV{<c>}{<q>} <Rd>, <Rm> {, <shift> #<amount>}
```

### MOVS, rotate right with extend variant

Applies when **S == 1** && **imm3 == 000** && **imm2 == 00** && **sr\_type == 11**.

MOVS{<c>}{<q>} <Rd>, <Rm>, RRX

### MOVS, shift or rotate by value variant

Applies when **S == 1** && **!(imm3 == 000 && imm2 == 00 && sr\_type == 11)**.

```
MOVS.W <Rd>, <Rm> {, <shift> #<amount>}
// Outside IT block
// and <Rd>, <Rm>, <shift>, <amount> can be represented in T1 or T2
MOVS{<c>}{<q>} <Rd>, <Rm> {, <shift> #<amount>}
```

### Decode for this encoding

```
1 if !HaveMainExt() then UNDEFINED;
2 d = UInt(Rd); m = UInt(Rm); setflags = (S == '1');
3 (shift_t, shift_n) = DecodeImmShift(sr_type, imm3:imm2);
4 if !setflags && (imm3:imm2:sr_type == '0000000') then
5 if (d == 15 || m == 15 || (d == 13 && m == 13)) then UNPREDICTABLE;
6 else
7 if (d IN {13,15} || m IN {13,15}) then UNPREDICTABLE;
```

### Alias conditions

| Alias            | variant | preferred when                                               |
|------------------|---------|--------------------------------------------------------------|
| ASRS (immediate) | T3      | S == '1' && sr_type == '10'                                  |
| ASRS (immediate) | T2      | op == '10' && !InITBlock()                                   |
| ASR (immediate)  | T3      | S == '0' && sr_type == '10'                                  |
| ASR (immediate)  | T2      | op == '10' && InITBlock()                                    |
| LSLS (immediate) | T3      | S == '1' && imm3:Rd:imm2 != '000xxxx00' && sr_type == '00'   |
| LSLS (immediate) | T2      | op == '00' && imm5 != '00000' && !InITBlock()                |
| LSL (immediate)  | T3      | S == '0' && imm3:Rd:imm2 != '000xxxx00' && sr_type == '00'   |
| LSL (immediate)  | T2      | op == '00' && imm5 != '00000' && InITBlock()                 |
| LSRS (immediate) | T3      | S == '1' && sr_type == '01'                                  |
| LSRS (immediate) | T2      | op == '01' && !InITBlock()                                   |
| LSR (immediate)  | T3      | S == '0' && sr_type == '01'                                  |
| LSR (immediate)  | T2      | op == '01' && InITBlock()                                    |
| RORS (immediate) | -       | S == '1' && imm3:Rd:imm2 != '000xxxx00' && sr_type == '11'   |
| ROR (immediate)  | -       | S == '0' && imm3:Rd:imm2 != '000xxxx00' && sr_type == '11'   |
| RRXS             | -       | S == '1' && imm3 == '000' && imm2 == '00' && sr_type == '11' |
| RRX              | -       | S == '0' && imm3 == '000' && imm2 == '00' && sr_type == '11' |

### Assembler symbols for all encodings

|          |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                 |
|----------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <c>      | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                        |
| <q>      | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                        |
| <Rd>     | For encoding T1: is the general-purpose destination register, encoded in the "D:Rd" field. If the PC is used: <ul style="list-style-type: none"> <li>- The instruction causes a simple branch to the address moved to the PC.</li> <li>- The instruction must either be outside an IT block or the last instruction of an IT block.</li> </ul> For encoding T2 and T3: is the general-purpose destination register, encoded in the "Rd" field.                                                                                                                                                                                                                                                  |
| <Rm>     | For encoding T1: is the general-purpose source register, encoded in the "Rm" field. The PC can be used.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |
| <shift>  | For encoding T2 and T3: is the general-purpose source register, encoded in the "Rm" field.<br>For encoding T2: is the type of shift to be applied to the source register, encoded in the "op" field. It can have the following values: <ul style="list-style-type: none"> <li>LSL when sr_type = 00</li> <li>LSR when sr_type = 01</li> <li>ASR when sr_type = 10</li> </ul> For encoding T3: is the type of shift to be applied to the source register, encoded in the "sr_type" field. It can have the following values: <ul style="list-style-type: none"> <li>LSL when sr_type = 00</li> <li>LSR when sr_type = 01</li> <li>ASR when sr_type = 10</li> <li>ROR when sr_type = 11</li> </ul> |
| <amount> | For encoding T2: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR) encoded in the "imm5" field as <amount> modulo 32.<br>For encoding T3: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR) encoded in the "imm3:imm2" field as <amount> modulo 32.                                                                                                                                                                                                                                                                                                                       |

### Operation for all encodings

```

1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 (result, carry) = Shift_C(R[m], shift_t, shift_n, APSR.C);
4 if d == 15 then
5 ALUwritePC(result); // setflags is always FALSE here
6 else
7 RSPCheck[d] = result;
8 if setflags then
9 APSR.N = result[31];
10 APSR.Z = IsZeroBit(result);

```

Chapter C2. Instruction Specification

C2.4. Alphabetical list of instructions

```
11 APSR.C = carry;
12 // APSR.V unchanged
```

## C2.4.92 MOV, MOVS (register-shifted register)

Move (register-shifted register). Move (register-shifted register) copies a register-shifted register value to the destination register. It can optionally update the condition flags based on the value.

This instruction is used by the aliases [ASRS \(register\)](#), [ASR \(register\)](#), [LSLS \(register\)](#), [LSL \(register\)](#), [LSRS \(register\)](#), [LSR \(register\)](#), [RORS \(register\)](#), [ROR \(register\)](#).

### T1

Armv8-M

|    |    |    |    |    |    |    |   |   |    |   |   |     |   |   |   |
|----|----|----|----|----|----|----|---|---|----|---|---|-----|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8 | 7 | 6  | 5 | 4 | 3   | 2 | 1 | 0 |
| 0  | 1  | 0  | 0  | 0  | 0  | op |   |   | Rs |   |   | Rdm |   |   |   |

#### Arithmetic shift right variant

Applies when **op == 0100**.

```
MOV<c>{<q>} <Rdm>, <Rdm>, ASR <Rs>
// Inside IT block
MOVS{<q>} <Rdm>, <Rdm>, ASR <Rs>
// Outside IT block
```

#### Logical shift left variant

Applies when **op == 0010**.

```
MOV<c>{<q>} <Rdm>, <Rdm>, LSL <Rs>
// Inside IT block
MOVS{<q>} <Rdm>, <Rdm>, LSL <Rs>
// Outside IT block
```

#### Logical shift right variant

Applies when **op == 0011**.

```
MOV<c>{<q>} <Rdm>, <Rdm>, LSR <Rs>
// Inside IT block
MOVS{<q>} <Rdm>, <Rdm>, LSR <Rs>
// Outside IT block
```

#### Rotate right variant

Applies when **op == 0111**.

```
MOV<c>{<q>} <Rdm>, <Rdm>, ROR <Rs>
// Inside IT block
MOVS{<q>} <Rdm>, <Rdm>, ROR <Rs>
// Outside IT block
```

#### Decode for this encoding

```
1 if !(op IN {'0010', '0011', '0100', '0111'}) then SEE "Related encodings";
2 d = UInt(Rdm); m = UInt(Rdm); s = UInt(Rs);
3 setflags = !InITBlock(); shift_t = DecodeRegShift(op[2]:op[0]);
```

### T2

Armv8-M Main Extension only





LSR when `sr_type = 01`  
 ASR when `sr_type = 10`  
 ROR when `sr_type = 11`

<Rs> Is the general-purpose source register holding a shift amount in its bottom 8 bits, encoded in the "Rs" field.

### Operation for all encodings

```

1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 shift_n = UInt(R[s][7:0]);
4 (result, carry) = Shift_C(R[m], shift_t, shift_n, APSR.C);
5 R[d] = result;
6 if setflags then
7 APSR.N = result[31];
8 APSR.Z = IsZeroBit(result);
9 APSR.C = carry;
10 // APSR.V unchanged

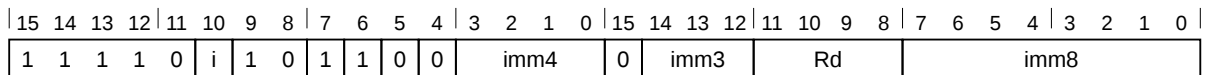
```

### C2.4.93 MOV<sub>T</sub>

Move Top. Move Top writes an immediate value to the top halfword of the destination register. It does not affect the contents of the bottom halfword.

#### T1

Armv8-M



#### T1 variant

MOV<sub>T</sub>{<c>}{<q>} <Rd>, #<imm16>

#### Decode for this encoding

```
1 d = UInt(Rd); imm16 = imm4:i:imm3:imm8;
2 if d IN {13,15} then UNPREDICTABLE;
```

#### Assembler symbols for all encodings

<c> See [C1.2.5 Standard assembler syntax fields](#) on page 309.  
 <q> See [C1.2.5 Standard assembler syntax fields](#) on page 309.  
 <Rd> Is the general-purpose destination register, encoded in the "Rd" field.  
 <imm16> Is a 16-bit unsigned immediate, in the range 0 to 65535, encoded in the "imm4:i:imm3:imm8" field.

#### Operation for all encodings

```
1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 R[d][31:16] = imm16;
4 // R[d][15:0] unchanged
```

## C2.4.94 MRC, MRC2

Move to Register from Coprocessor. Move to Register from Coprocessor causes a coprocessor to transfer a value to a general-purpose register or to the condition flags.

Arm reserves coprocessors CP8 to CP15, and this manual defines the valid instructions when coproc is in this range.

### T1

*Armv8-M Main Extension only*

|    |    |    |    |    |    |   |   |      |   |   |   |   |     |   |   |    |    |    |    |    |        |   |   |   |      |   |   |   |   |     |   |  |  |
|----|----|----|----|----|----|---|---|------|---|---|---|---|-----|---|---|----|----|----|----|----|--------|---|---|---|------|---|---|---|---|-----|---|--|--|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7    | 6 | 5 | 4 | 3 | 2   | 1 | 0 | 15 | 14 | 13 | 12 | 11 | 10     | 9 | 8 | 7 | 6    | 5 | 4 | 3 | 2 | 1   | 0 |  |  |
| 1  | 1  | 1  | 0  | 1  | 1  | 1 | 0 | opc1 |   |   |   | 1 | CRn |   |   |    | Rt |    |    |    | coproc |   |   |   | opc2 |   |   |   | 1 | CRm |   |  |  |

#### T1 variant

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

#### Decode for this encoding

```

1 if coproc IN '101x' then SEE "Floating-point";
2 if !HaveMainExt() then UNDEFINED;
3 t = UInt(Rt); cp = UInt(coproc);
4 if t == 13 then UNPREDICTABLE;

```

### T2

*Armv8-M Main Extension only*

|    |    |    |    |    |    |   |   |      |   |   |   |   |     |   |   |    |    |    |    |    |        |   |   |   |      |   |   |   |   |     |   |  |  |
|----|----|----|----|----|----|---|---|------|---|---|---|---|-----|---|---|----|----|----|----|----|--------|---|---|---|------|---|---|---|---|-----|---|--|--|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7    | 6 | 5 | 4 | 3 | 2   | 1 | 0 | 15 | 14 | 13 | 12 | 11 | 10     | 9 | 8 | 7 | 6    | 5 | 4 | 3 | 2 | 1   | 0 |  |  |
| 1  | 1  | 1  | 1  | 1  | 1  | 1 | 0 | opc1 |   |   |   | 1 | CRn |   |   |    | Rt |    |    |    | coproc |   |   |   | opc2 |   |   |   | 1 | CRm |   |  |  |

#### T2 variant

MRC2{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

#### Decode for this encoding

```

1 if coproc IN '101x' then UNDEFINED;
2 if !HaveMainExt() then UNDEFINED;
3 t = UInt(Rt); cp = UInt(coproc);
4 if t == 13 then UNPREDICTABLE;

```

#### Assembler symbols for all encodings

|          |                                                                                                                                                                                                                                               |
|----------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <c>      | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                                                                                                                                                                      |
| <q>      | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                                                                                                                                                                      |
| <coproc> | Is the name of the coprocessor, encoded in the "coproc" field. The generic coprocessor names are p0 to p15.                                                                                                                                   |
| <opc1>   | Is a coprocessor-specific opcode in the range 0 to 7, encoded in the "opc1" field.                                                                                                                                                            |
| <Rt>     | Is the general-purpose register to be transferred or <code>APSR_nzcv</code> (encoded as 0b1111), encoded in the "Rt" field. If <code>APSR_nzcv</code> is used, bits [31:28] of the transferred value are written to the APSR condition flags. |
| <CRn>    | Is the first coprocessor register, encoded in the "CRn" field.                                                                                                                                                                                |
| <CRm>    | Is the second coprocessor register, encoded in the "CRm" field.                                                                                                                                                                               |
| <opc2>   | Is a coprocessor-specific opcode in the range 0 to 7, defaulting to 0 and encoded in the "opc2" field.                                                                                                                                        |

### Operation for all encodings

```
1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 ExecuteCPCheck(cp);
4 if !Coprocc_Accepted(cp, ThisInstr()) then
5 GenerateCoproccException();
6 else
7 value = Coproc_GetOneWord(cp, ThisInstr());
8 if t != 15 then
9 R[t] = value;
10 else
11 APSR.N = value[31];
12 APSR.Z = value[30];
13 APSR.C = value[29];
14 APSR.V = value[28];
15 // value[27:0] are not used.
```

### C2.4.95 MRRC, MRRC2

Move to two Registers from Coprocessor. Move to two Registers from Coprocessor causes a coprocessor to transfer values to two general-purpose registers.

If no coprocessor can execute the instruction, a UsageFault exception is generated.

Arm reserves coprocessors CP8 to CP15, and this manual defines the valid instructions when coproc is in this range.

#### T1

Armv8-M Main Extension only

|    |    |    |    |    |    |   |   |   |   |   |   |     |   |   |   |    |    |    |    |        |    |   |   |      |   |   |   |     |   |   |   |
|----|----|----|----|----|----|---|---|---|---|---|---|-----|---|---|---|----|----|----|----|--------|----|---|---|------|---|---|---|-----|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3   | 2 | 1 | 0 | 15 | 14 | 13 | 12 | 11     | 10 | 9 | 8 | 7    | 6 | 5 | 4 | 3   | 2 | 1 | 0 |
| 1  | 1  | 1  | 0  | 1  | 1  | 0 | 0 | 0 | 1 | 0 | 1 | Rt2 |   |   |   | Rt |    |    |    | coproc |    |   |   | opc1 |   |   |   | CRm |   |   |   |

#### T1 variant

MRRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <Rt2>, <CRm>

#### Decode for this encoding

```

1 if coproc IN '101x' then SEE "Floating-point";
2 if !HaveMainExt() then UNDEFINED;
3 t = UInt(Rt); t2 = UInt(Rt2); cp = UInt(coproc);
4 if t == 15 || t2 == 15 || t == t2 then UNPREDICTABLE;
5 if t == 13 || t2 == 13 then UNPREDICTABLE;

```

#### CONSTRAINED UNPREDICTABLE behavior

If  $t == t2$ , then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The value in the destination register is UNKNOWN.

#### T2

Armv8-M Main Extension only

|    |    |    |    |    |    |   |   |   |   |   |   |     |   |   |   |    |    |    |    |        |    |   |   |      |   |   |   |     |   |   |   |
|----|----|----|----|----|----|---|---|---|---|---|---|-----|---|---|---|----|----|----|----|--------|----|---|---|------|---|---|---|-----|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3   | 2 | 1 | 0 | 15 | 14 | 13 | 12 | 11     | 10 | 9 | 8 | 7    | 6 | 5 | 4 | 3   | 2 | 1 | 0 |
| 1  | 1  | 1  | 1  | 1  | 1  | 0 | 0 | 0 | 1 | 0 | 1 | Rt2 |   |   |   | Rt |    |    |    | coproc |    |   |   | opc1 |   |   |   | CRm |   |   |   |

#### T2 variant

MRRC2{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <Rt2>, <CRm>

#### Decode for this encoding

```

1 if coproc IN '101x' then UNDEFINED;
2 if !HaveMainExt() then UNDEFINED;
3 t = UInt(Rt); t2 = UInt(Rt2); cp = UInt(coproc);
4 if t == 15 || t2 == 15 || t == t2 then UNPREDICTABLE;
5 if t == 13 || t2 == 13 then UNPREDICTABLE;

```

#### CONSTRAINED UNPREDICTABLE behavior

If  $t == t2$ , then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The value in the destination register is UNKNOWN.

### Assembler symbols for all encodings

|          |                                                                                                             |
|----------|-------------------------------------------------------------------------------------------------------------|
| <c>      | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                                    |
| <q>      | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                                    |
| <coproc> | Is the name of the coprocessor, encoded in the "coproc" field. The generic coprocessor names are p0 to p15. |
| <opc1>   | Is a coprocessor-specific opcode in the range 0 to 15, encoded in the "opc1" field.                         |
| <Rt>     | Is the first general-purpose register to be transferred, encoded in the "Rt" field.                         |
| <Rt2>    | Is the second general-purpose register to be transferred, encoded in the "Rt2" field.                       |
| <CRm>    | Is a coprocessor register, encoded in the "CRm" field.                                                      |

### Operation for all encodings

```
1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 ExecuteCPCheck(cp);
4 if !Coprocc_Accepted(cp, ThisInstr()) then
5 GenerateCoproccorException();
6 else
7 (R[t2], R[t]) = Coproc_GetTwoWords(cp, ThisInstr());
```

## C2.4.96 MRS

Move to Register from Special register. Move to Register from Special register moves the value from the selected special-purpose register into a general-purpose register.

### T1

Armv8-M

|    |    |    |    |    |    |   |   |   |   |   |     |     |     |     |     |    |    |     |    |    |    |   |   |   |   |   |   |      |   |   |   |  |  |  |  |
|----|----|----|----|----|----|---|---|---|---|---|-----|-----|-----|-----|-----|----|----|-----|----|----|----|---|---|---|---|---|---|------|---|---|---|--|--|--|--|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4   | 3   | 2   | 1   | 0   | 15 | 14 | 13  | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3    | 2 | 1 | 0 |  |  |  |  |
| 1  | 1  | 1  | 1  | 0  | 0  | 1 | 1 | 1 | 1 | 1 | (0) | (1) | (1) | (1) | (1) | 1  | 0  | (0) | 0  | Rd |    |   |   |   |   |   |   | SYSm |   |   |   |  |  |  |  |

### T1 variant

MRS{<c>}{<q>} <Rd>, <spec\_reg>

### Decode for this encoding

```
1 d = UInt(Rd);
2 if d IN {13,15} then UNPREDICTABLE;
```

### Assembler symbols for all encodings

<c> See [C1.2.5 Standard assembler syntax fields](#) on page 309.  
 <q> See [C1.2.5 Standard assembler syntax fields](#) on page 309.  
 <Rd> Is the general-purpose destination register, encoded in the "Rd" field.  
 <spec\_reg> Is the special register to be accessed, encoded in the "SYSm" field. It can have the following values:

|              |                      |
|--------------|----------------------|
| APSR         | when SYSm = 00000000 |
| IAPSR        | when SYSm = 00000001 |
| EAPSR        | when SYSm = 00000010 |
| XPSR         | when SYSm = 00000011 |
| IPSR         | when SYSm = 00000101 |
| EPSR         | when SYSm = 00000110 |
| IEPSR        | when SYSm = 00000111 |
| MSP          | when SYSm = 00001000 |
| PSP          | when SYSm = 00001001 |
| MSPLIM       | when SYSm = 00001010 |
| PSPLIM       | when SYSm = 00001011 |
| PRIMASK      | when SYSm = 00010000 |
| BASEPRI      | when SYSm = 00010001 |
| BASEPRI_MAX  | when SYSm = 00010010 |
| FAULTMASK    | when SYSm = 00010011 |
| CONTROL      | when SYSm = 00010100 |
| MSP_NS       | when SYSm = 10001000 |
| PSP_NS       | when SYSm = 10001001 |
| MSPLIM_NS    | when SYSm = 10001010 |
| PSPLIM_NS    | when SYSm = 10001011 |
| PRIMASK_NS   | when SYSm = 10010000 |
| BASEPRI_NS   | when SYSm = 10010001 |
| FAULTMASK_NS | when SYSm = 10010011 |
| CONTROL_NS   | when SYSm = 10010100 |
| SP_NS        | when SYSm = 10011000 |

The following encodings are UNPREDICTABLE:

- SYSm = 00000100
- SYSm = 000011xx

```
- SYSm = 00010101
- SYSm = 0001011x
- SYSm = 00011xxx
- SYSm = 001xxxxx
- SYSm = 01xxxxxx
- SYSm = 10000xxx
- SYSm = 100011xx
- SYSm = 10010010
- SYSm = 10010101
- SYSm = 1001011x
- SYSm = 10011001
- SYSm = 1001101x
- SYSm = 100111xx
- SYSm = 101xxxxx
- SYSm = 11xxxxxx
```

An access to a register not ending in `_NS` returns the register associated with the current Security state. Access to a register ending in `_NS` in Secure state returns the Non-secure register. Access to a register ending in `_NS` in Non-secure state is RAZ/WI. Access to `BASEPRI_MAX` returns the contents of `BASEPRI`.

### Operation for all encodings

```
1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 R[d] = Zeros(32);
4
5 // NOTE: the MSB of SYSm is used to select between either the current
6 // domains view of the registers and other domains view of the register.
7 // This is required so that the Secure state can access the Non-secure
8 // versions of banked registers. For security reasons the Secure versions of
9 // the registers are not accessible from the Non-secure state.
10 case SYSm[7:3] of
11 when '00000' /* XPSR accesses */
12 if UInt(SYSm) == 4 then UNPREDICTABLE;
13 if CurrentModeIsPrivileged() && SYSm[0] == '1' then
14 R[d][8:0] = IPSR.Exception;
15 if SYSm[1] == '1' then
16 R[d][26:24] = '000'; /* EPSR reads as zero */
17 R[d][15:10] = '000000';
18 if SYSm[2] == '0' then
19 R[d][31:27] = APSR[31:27];
20 if HaveDSPExt() then
21 R[d][19:16] = APSR[19:16];
22 when '00001' /* SP access */
23 if CurrentModeIsPrivileged() then
24 case SYSm[2:0] of
25 when '000'
26 R[d] = SP_Main;
27 when '001'
28 R[d] = SP_Process;
29 when '010'
30 if IsSecure() then
31 R[d] = MSPLIM_S.LIMIT:'000';
32 else
33 if HaveMainExt() then
34 R[d] = MSPLIM_NS.LIMIT:'000';
35 else
36 UNPREDICTABLE;
37 when '011'
38 if IsSecure() then
39 R[d] = PSPLIM_S.LIMIT:'000';
40 else
41 if HaveMainExt() then
42 R[d] = PSPLIM_NS.LIMIT:'000';
```



```

43 else
44 UNPREDICTABLE;
45 otherwise
46 UNPREDICTABLE;
47 when '10001' /* SP access - alt domain */
48 if !HaveSecurityExt() then UNPREDICTABLE;
49 if CurrentModeIsPrivileged() && CurrentState == SecurityState_Secure then
50 case SYSm[2:0] of
51 when '000'
52 R[d] = SP_Main_NonSecure;
53 when '001'
54 R[d] = SP_Process_NonSecure;
55 when '010'
56 if HaveMainExt() then
57 R[d] = MSPLIM_NS.LIMIT:'000';
58 else
59 UNPREDICTABLE;
60 when '011'
61 if HaveMainExt() then
62 R[d] = PSPLIM_NS.LIMIT:'000';
63 else
64 UNPREDICTABLE;
65 otherwise
66 UNPREDICTABLE;
67 when '00010' /* Priority mask or CONTROL access */
68 case SYSm[2:0] of
69 when '000'
70 if CurrentModeIsPrivileged() then
71 R[d][0] = PRIMASK.PM;
72 when '001'
73 if HaveMainExt() then
74 if CurrentModeIsPrivileged() then
75 R[d][7:0] = BASEPRI[7:0];
76 else
77 UNPREDICTABLE;
78 when '010'
79 if HaveMainExt() then
80 if CurrentModeIsPrivileged() then
81 R[d][7:0] = BASEPRI[7:0];
82 else
83 UNPREDICTABLE;
84 when '011'
85 if HaveMainExt() then
86 if CurrentModeIsPrivileged() then
87 R[d][0] = FAULTMASK.FM;
88 else
89 UNPREDICTABLE;
90 when '100'
91 if HaveFPEExt() && IsSecure() then
92 R[d][3:0] = CONTROL[3:0];
93 elsif HaveFPEExt() then
94 R[d][2:0] = CONTROL[2:0];
95 else
96 R[d][1:0] = CONTROL[1:0];
97 otherwise
98 UNPREDICTABLE;
99 when '10010' /* Priority mask or CONTROL access - alt
100 domain */
101 if !HaveSecurityExt() then UNPREDICTABLE;
102 if CurrentState == SecurityState_Secure then
103 case SYSm[2:0] of
104 when '000'
105 if CurrentModeIsPrivileged() then
106 R[d][0] = PRIMASK_NS.PM;
107 when '001'
108 if HaveMainExt() then
109 if CurrentModeIsPrivileged() then
110 R[d][7:0] = BASEPRI_NS[7:0];
111 else

```

```
111 UNPREDICTABLE;
112 when '011'
113 if HaveMainExt() then
114 if CurrentModeIsPrivileged() then
115 R[d][0] = FAULTMASK_NS.FM;
116 else
117 UNPREDICTABLE;
118 when '100'
119 if HaveFPExt() then
120 R[d][2:0] = CONTROL_NS[2:0];
121 else
122 R[d][1:0] = CONTROL_NS[1:0];
123 otherwise
124 UNPREDICTABLE;
125 when '10011' /* SP_NS - Non-secure stack pointer */
126 if !HaveSecurityExt() then UNPREDICTABLE;
127 if CurrentState == SecurityState_Secure then
128 case SYSm[2:0] of
129 when '000'
130 R[d] = _SP(LookUpSP_with_security_mode(FALSE, CurrentMode()));
131 otherwise
132 UNPREDICTABLE;
133 otherwise
134 UNPREDICTABLE;
```

### CONSTRAINED UNPREDICTABLE behavior

If SYSm not valid special register, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The value in the destination register is UNKNOWN.



|              |                      |
|--------------|----------------------|
| PRIMASK      | when SYSm = 00010000 |
| BASEPRI      | when SYSm = 00010001 |
| BASEPRI_MAX  | when SYSm = 00010010 |
| FAULTMASK    | when SYSm = 00010011 |
| CONTROL      | when SYSm = 00010100 |
| MSP_NS       | when SYSm = 10001000 |
| PSP_NS       | when SYSm = 10001001 |
| MSPLIM_NS    | when SYSm = 10001010 |
| PSPLIM_NS    | when SYSm = 10001011 |
| PRIMASK_NS   | when SYSm = 10010000 |
| BASEPRI_NS   | when SYSm = 10010001 |
| FAULTMASK_NS | when SYSm = 10010011 |
| CONTROL_NS   | when SYSm = 10010100 |
| SP_NS        | when SYSm = 10011000 |

The following encodings are UNPREDICTABLE:

- SYSm = 00000100
- SYSm = 000011xx
- SYSm = 00010101
- SYSm = 0001011x
- SYSm = 00011xxx
- SYSm = 001xxxxx
- SYSm = 01xxxxxx
- SYSm = 10000xxx
- SYSm = 100011xx
- SYSm = 10010010
- SYSm = 10010101
- SYSm = 1001011x
- SYSm = 10011001
- SYSm = 1001101x
- SYSm = 100111xx
- SYSm = 101xxxxx
- SYSm = 11xxxxxx

An access to a register not ending in `_NS` returns the register associated with the current Security state. Access to a register ending in `_NS` in Secure state returns the Non-secure register. Access to a register ending in `_NS` in Non-secure state is RAZ/WI. Access to `BASEPRI_MAX` writes to `BASEPRI` if the priority that is written is higher than the existing priority in `BASEPRI`. Otherwise, the access is ignored.

<Rn>      Is the general-purpose source register, encoded in the "Rn" field.

### Operation for all encodings

```

1 if ConditionPassed() then
2 EncodingSpecificOperations();
3
4 // NOTE: the MSB of SYSm is used to select between either the current
5 // domains view of the registers and other domains view of the register.
6 // This is required to that the Secure state can access the Non-secure
7 // versions of banked registers. For security reasons the Secure versions of
8 // the registers are not accessible from the Non-secure state.
9 case SYSm[7:3] of
10 when '00000'
11 if UInt(SYSm) == 4 then UNPREDICTABLE;
12 if SYSm[2] == '0' then
13 if mask[0] == '1' then
14 if !HaveDSPExt() then
15 UNPREDICTABLE;
16 else
17 APSR[19:16] = R[n][19:16];
18 if mask[1] == '1' then
19 APSR[31:27] = R[n][31:27];

```

```

20 when '00001' /* SP access */
21 if CurrentModeIsPrivileged() then
22 case SYSm[2:0] of
23 when '000'
24 // MSR not subject to SP limit, write directly to register.
25 if IsSecure() then
26 _R[RNameSP_Main_Secure] = R[n][31:2]:'00';
27 else
28 _R[RNameSP_Main_NonSecure] = R[n][31:2]:'00';
29 when '001'
30 // MSR not subject to SP limit, write directly to register.
31 if IsSecure() then
32 _R[RNameSP_Process_Secure] = R[n][31:2]:'00';
33 else
34 _R[RNameSP_Process_NonSecure] = R[n][31:2]:'00';
35 when '010'
36 if IsSecure() then
37 MSPLIM_S.LIMIT = R[n][31:3];
38 else
39 if HaveMainExt() then
40 MSPLIM_NS.LIMIT = R[n][31:3];
41 else
42 UNPREDICTABLE;
43 when '011'
44 if IsSecure() then
45 PSPLIM_S.LIMIT = R[n][31:3];
46 else
47 if HaveMainExt() then
48 PSPLIM_NS.LIMIT = R[n][31:3];
49 else
50 UNPREDICTABLE;
51 otherwise
52 UNPREDICTABLE;
53 when '10001' /* SP access - alt domain */
54 if !HaveSecurityExt() then UNPREDICTABLE;
55 if CurrentModeIsPrivileged() && CurrentState == SecurityState_Secure then
56 case SYSm[2:0] of
57 when '000'
58 // MSR not subject to SP limit, write directly to register.
59 _R[RNameSP_Main_NonSecure] = R[n][31:2]:'00';
60 when '001'
61 // MSR not subject to SP limit, write directly to register.
62 _R[RNameSP_Process_NonSecure] = R[n][31:2]:'00';
63 when '010'
64 if HaveMainExt() then
65 MSPLIM_NS.LIMIT = R[n][31:3];
66 else
67 UNPREDICTABLE;
68 when '011'
69 if HaveMainExt() then
70 PSPLIM_NS.LIMIT = R[n][31:3];
71 else
72 UNPREDICTABLE;
73 otherwise
74 UNPREDICTABLE;
75 when '00010' /* Priority mask or CONTROL access */
76 case SYSm[2:0] of
77 when '000'
78 if CurrentModeIsPrivileged() then
79 PRIMASK.PM = R[n][0];
80 when '001'
81 if CurrentModeIsPrivileged() then
82 if HaveMainExt() then
83 BASEPRI[7:0] = R[n][7:0];
84 else
85 UNPREDICTABLE;
86 when '010'
87 if CurrentModeIsPrivileged() then
88 if HaveMainExt() then

```

```

89 if (R[n][7:0] != '00000000') &&
90 (UInt(R[n][7:0]) < UInt(BASEPRI[7:0]) || BASEPRI[7:0] == '
 00000000') then
91 BASEPRI[7:0] = R[n][7:0];
92 else
93 UNPREDICTABLE;
94 when '011'
95 if CurrentModeIsPrivileged() then
96 if HaveMainExt() then
97 if ExecutionPriority() > -1 || R[n][0] == '0' then
98 FAULTMASK.FM = R[n][0];
99 else
100 UNPREDICTABLE;
101 when '100'
102 if CurrentModeIsPrivileged() then
103 CONTROL.nPRIV = R[n][0];
104 CONTROL.SPSEL = R[n][1];
105 if HaveFPExt() && (IsSecure() || NSACR.CP10 == '1') then
106 CONTROL.FPCA = R[n][2];
107 if HaveFPExt() && IsSecure() then
108 CONTROL_S.SFPA = R[n][3];
109 otherwise
110 UNPREDICTABLE;
111 when '10010' /* Priority mask or CONTROL access - alt
 domain */
112 if !HaveSecurityExt() then UNPREDICTABLE;
113 if CurrentModeIsPrivileged() && CurrentState == SecurityState_Secure then
114 case SYSm[2:0] of
115 when '000'
116 PRIMASK_NS.PM = R[n][0];
117 when '001'
118 if HaveMainExt() then
119 BASEPRI_NS[7:0] = R[n][7:0];
120 else
121 UNPREDICTABLE;
122 when '011'
123 if HaveMainExt() then
124 if ExecutionPriority() > -1 || R[n][0] == '0' then
125 FAULTMASK_NS.FM = R[n][0];
126 else
127 UNPREDICTABLE;
128 when '100'
129 CONTROL_NS.nPRIV = R[n][0];
130 CONTROL_NS.SPSEL = R[n][1];
131 if HaveFPExt() then CONTROL_NS.FPCA = R[n][2];
132 otherwise
133 UNPREDICTABLE;
134 when '10011' /* SP_NS - Non-secure stack pointer */
135 if !HaveSecurityExt() then UNPREDICTABLE;
136 if CurrentState == SecurityState_Secure then
137 case SYSm[2:0] of
138 when '000'
139 spName = LookUpSP_with_security_mode(FALSE, CurrentMode());
140 // MSR SP_NS is subject to SP limit check.
141 - = _SP(spName, FALSE, R[n]);
142 otherwise
143 UNPREDICTABLE;
144 otherwise
145 UNPREDICTABLE;

```

### CONSTRAINED UNPREDICTABLE behavior

If SYSm not valid special register, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.

- The instruction treats SYSm as UNKNOWN.

## C2.4.98 MUL

**Multiply.** Multiply multiplies two register values. The least significant 32 bits of the result are written to the destination register. These 32 bits do not depend on whether signed or unsigned calculations are performed.

It can optionally update the condition flags based on the result. In the T32 instruction set, this option is limited to only a few forms of the instruction. Use of this option adversely affects performance on many implementations.

### T1

*Armv8-M*

|    |    |    |    |    |    |   |   |   |   |    |   |   |     |   |   |
|----|----|----|----|----|----|---|---|---|---|----|---|---|-----|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5  | 4 | 3 | 2   | 1 | 0 |
| 0  | 1  | 0  | 0  | 0  | 0  | 1 | 1 | 0 | 1 | Rn |   |   | Rdm |   |   |

#### T1 variant

```
MUL<c>{<q>} <Rdm>, <Rn>{, <Rdm>}
// Inside IT block
MULS{<q>} <Rdm>, <Rn>{, <Rdm>}
// Outside IT block
```

#### Decode for this encoding

```
1 d = UInt(Rdm); n = UInt(Rn); m = UInt(Rdm); setflags = !InITBlock();
```

### T2

*Armv8-M Main Extension only*

|    |    |    |    |    |    |   |   |   |   |   |   |    |   |   |   |    |    |    |    |    |    |   |   |   |   |    |   |   |   |   |   |
|----|----|----|----|----|----|---|---|---|---|---|---|----|---|---|---|----|----|----|----|----|----|---|---|---|---|----|---|---|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3  | 2 | 1 | 0 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5  | 4 | 3 | 2 | 1 | 0 |
| 1  | 1  | 1  | 1  | 1  | 0  | 1 | 1 | 0 | 0 | 0 | 0 | Rn |   |   | 1 | 1  | 1  | 1  | Rd |    |    | 0 | 0 | 0 | 0 | Rm |   |   |   |   |   |

#### T2 variant

```
MUL<c>.W <Rd>, <Rn>{, <Rm>}
// Inside IT block, and <Rd>, <Rn>, <Rm> can be represented in T1
MUL{<c>}{<q>} <Rd>, <Rn>{, <Rm>}
```

#### Decode for this encoding

```
1 if !HaveMainExt() then UNDEFINED;
2 d = UInt(Rd); n = UInt(Rn); m = UInt(Rm); setflags = FALSE;
3 if d IN {13,15} || n IN {13,15} || m IN {13,15} then UNPREDICTABLE;
```

#### Assembler symbols for all encodings

|       |                                                                                                                                |
|-------|--------------------------------------------------------------------------------------------------------------------------------|
| <c>   | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                                                       |
| <q>   | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                                                       |
| <Rdm> | Is the second general-purpose source register holding the multiplier and the destination register, encoded in the "Rdm" field. |
| <Rd>  | Is the general-purpose destination register, encoded in the "Rd" field.                                                        |
| <Rn>  | Is the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.                              |
| <Rm>  | Is the second general-purpose source register holding the multiplier, encoded in the "Rm" field. If omitted, <Rd> is used.     |



**Operation for all encodings**

```
1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 operand1 = SInt(R[n]); // operand1 = UInt(R[n]) produces the same final results
4 operand2 = SInt(R[m]); // operand2 = UInt(R[m]) produces the same final results
5 result = operand1 * operand2;
6 R[d] = result[31:0];
7 if setflags then
8 APSR.N = result[31];
9 APSR.Z = IsZeroBit(result[31:0]);
10 // APSR.C unchanged
11 // APSR.V unchanged
```

### C2.4.99 MVN (immediate)

Bitwise NOT (immediate). Bitwise NOT (immediate) writes the bitwise inverse of an immediate value to the destination register. It can optionally update the condition flags based on the value.

#### T1

Armv8-M Main Extension only

|    |    |    |    |    |    |   |   |   |   |   |   |   |   |   |   |    |      |    |    |    |    |   |   |      |   |   |   |   |   |   |   |
|----|----|----|----|----|----|---|---|---|---|---|---|---|---|---|---|----|------|----|----|----|----|---|---|------|---|---|---|---|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | 15 | 14   | 13 | 12 | 11 | 10 | 9 | 8 | 7    | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 1  | 1  | 1  | 1  | 0  | i  | 0 | 0 | 0 | 1 | 1 | S | 1 | 1 | 1 | 1 | 0  | imm3 | Rd |    |    |    |   |   | imm8 |   |   |   |   |   |   |   |

#### MVN variant

Applies when **S == 0**.

MVN{<c>}{<q>} <Rd>, #<const>

#### MVNS variant

Applies when **S == 1**.

MVNS{<c>}{<q>} <Rd>, #<const>

#### Decode for this encoding

```

1 if !HaveMainExt() then UNDEFINED;
2 d = UInt(Rd); setflags = (S == '1');
3 (imm32, carry) = T32ExpandImm_C(i:imm3:imm8, APSR.C);
4 if d IN {13,15} then UNPREDICTABLE;

```

#### Assembler symbols for all encodings

<c> See [C1.2.5 Standard assembler syntax fields](#) on page 309.  
 <q> See [C1.2.5 Standard assembler syntax fields](#) on page 309.  
 <Rd> Is the general-purpose destination register, encoded in the "Rd" field.  
 <const> Is an immediate value derived from the 12-bit immediate that is encoded in the 'i:imm3:imm8' field. See [C1.5 Modified immediate constants](#) on page 324 for the range of values.

#### Operation for all encodings

```

1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 result = NOT(imm32);
4 R[d] = result;
5 if setflags then
6 APSR.N = result[31];
7 APSR.Z = IsZeroBit(result);
8 APSR.C = carry;
9 // APSR.V unchanged

```



### Decode for this encoding

```
1 if !HaveMainExt() then UNDEFINED;
2 d = UInt(Rd); m = UInt(Rm); setflags = (S == '1');
3 (shift_t, shift_n) = DecodeImmShift(sr_type, imm3:imm2);
4 if d IN {13,15} || m IN {13,15} then UNPREDICTABLE;
```

### Assembler symbols for all encodings

<c> See [C1.2.5 Standard assembler syntax fields](#) on page 309.

<q> See [C1.2.5 Standard assembler syntax fields](#) on page 309.

<Rd> Is the general-purpose destination register, encoded in the "Rd" field.

<Rm> Is the general-purpose source register, encoded in the "Rm" field.

<shift> Is the type of shift to be applied to the source register, encoded in the "sr\_type" field. It can have the following values:

- LSL when sr\_type = 00
- LSR when sr\_type = 01
- ASR when sr\_type = 10
- ROR when sr\_type = 11

<amount> Is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR) encoded in the "imm3:imm2" field as <amount> modulo 32.

### Operation for all encodings

```
1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 (shifted, carry) = Shift_C(R[m], shift_t, shift_n, APSR.C);
4 result = NOT(shifted);
5 R[d] = result;
6 if setflags then
7 APSR.N = result[31];
8 APSR.Z = IsZeroBit(result);
9 APSR.C = carry;
10 // APSR.V unchanged
```

## C2.4.101 NOP

No Operation. No Operation does nothing.

The architecture makes no guarantees about any timing effects of including a NOP instruction.

This is a NOP-compatible hint.

### T1

*Armv8-M*

|    |    |    |    |    |    |   |   |   |   |   |   |   |   |   |   |
|----|----|----|----|----|----|---|---|---|---|---|---|---|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 1  | 0  | 1  | 1  | 1  | 1  | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

#### T1 variant

NOP {<c>} {<q>}

#### Decode for this encoding

```
1 // No additional decoding required
```

### T2

*Armv8-M Main Extension only*

|    |    |    |    |    |    |   |   |   |   |   |   |     |     |     |     |    |    |     |    |     |    |   |   |   |   |   |   |   |   |   |   |
|----|----|----|----|----|----|---|---|---|---|---|---|-----|-----|-----|-----|----|----|-----|----|-----|----|---|---|---|---|---|---|---|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3   | 2   | 1   | 0   | 15 | 14 | 13  | 12 | 11  | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 1  | 1  | 1  | 1  | 0  | 0  | 1 | 1 | 1 | 0 | 1 | 0 | (1) | (1) | (1) | (1) | 1  | 0  | (0) | 0  | (0) | 0  | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |   |

#### T2 variant

NOP {<c>} .W

#### Decode for this encoding

```
1 if !HaveMainExt() then UNDEFINED;
2 // No additional decoding required
```

#### Assembler symbols for all encodings

<c> See [C1.2.5 Standard assembler syntax fields](#) on page 309.  
 <q> See [C1.2.5 Standard assembler syntax fields](#) on page 309.

#### Operation for all encodings

```
1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 // Do nothing
```

### C2.4.102 ORN (immediate)

Logical OR NOT (immediate). Logical OR NOT (immediate) performs a bitwise (inclusive) OR of a register value and the complement of an immediate value, and writes the result to the destination register. It can optionally update the condition flags based on the result.

#### T1

Armv8-M Main Extension only

|    |    |    |    |    |    |   |   |   |   |   |   |            |   |      |    |      |    |    |    |    |    |   |   |   |   |   |   |   |   |   |   |
|----|----|----|----|----|----|---|---|---|---|---|---|------------|---|------|----|------|----|----|----|----|----|---|---|---|---|---|---|---|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3          | 2 | 1    | 0  | 15   | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 1  | 1  | 1  | 1  | 0  | i  | 0 | 0 | 0 | 1 | 1 | S | Rn != 1111 | 0 | imm3 | Rd | imm8 |    |    |    |    |    |   |   |   |   |   |   |   |   |   |   |

#### Flag setting variant

Applies when **S == 1**.

ORNS{<c>}{<q>} {<Rd>}, {<Rn>}, #<const>

#### Non flag setting variant

Applies when **S == 0**.

ORN{<c>}{<q>} {<Rd>}, {<Rn>}, #<const>

#### Decode for this encoding

```

1 if Rn == '1111' then SEE "MVN (immediate)";
2 if !HaveMainExt() then UNDEFINED;
3 d = UInt(Rd); n = UInt(Rn); setflags = (S == '1');
4 (imm32, carry) = T32ExpandImm_C(i:imm3:imm8, APSR.C);
5 if d IN {13,15} || n == 13 then UNPREDICTABLE;

```

#### Assembler symbols for all encodings

- <c> See [C1.2.5 Standard assembler syntax fields](#) on page 309.
- <q> See [C1.2.5 Standard assembler syntax fields](#) on page 309.
- <Rd> Is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as <Rn>.
- <Rn> Is the general-purpose source register, encoded in the "Rn" field.
- <const> Is an immediate value derived from the 12-bit immediate that is encoded in the 'i:imm3:imm8' field. See [C1.5 Modified immediate constants](#) on page 324 for the range of values.

#### Operation for all encodings

```

1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 result = R[n] OR NOT(imm32);
4 R[d] = result;
5 if setflags then
6 APSR.N = result[31];
7 APSR.Z = IsZeroBit(result);
8 APSR.C = carry;
9 // APSR.V unchanged

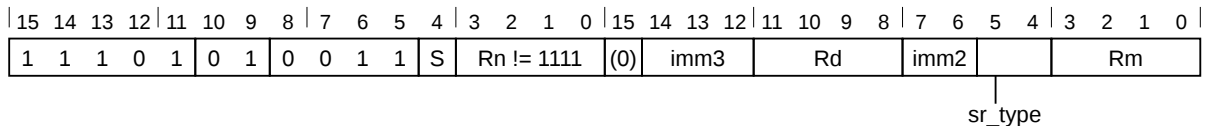
```

### C2.4.103 ORN (register)

Logical OR NOT (register). Logical OR NOT (register) performs a bitwise (inclusive) OR of a register value and the complement of an optionally-shifted register value, and writes the result to the destination register. It can optionally update the condition flags based on the result.

#### T1

Armv8-M Main Extension only



#### ORN, rotate right with extend variant

Applies when **S == 0** && **imm3 == 000** && **imm2 == 00** && **sr\_type == 11**.

ORN{<c>}{<q>} {<Rd>}, <Rn>, <Rm>, RRX

#### ORN, shift or rotate by value variant

Applies when **S == 0** && **!(imm3 == 000 && imm2 == 00 && sr\_type == 11)**.

ORN{<c>}{<q>} {<Rd>}, <Rn>, <Rm> {, <shift> #<amount>}

#### ORNS, rotate right with extend variant

Applies when **S == 1** && **imm3 == 000** && **imm2 == 00** && **sr\_type == 11**.

ORNS{<c>}{<q>} {<Rd>}, <Rn>, <Rm>, RRX

#### ORNS, shift or rotate by value variant

Applies when **S == 1** && **!(imm3 == 000 && imm2 == 00 && sr\_type == 11)**.

ORNS{<c>}{<q>} {<Rd>}, <Rn>, <Rm> {, <shift> #<amount>}

#### Decode for this encoding

```

1 if Rn == '1111' then SEE "MVN (register)";
2 if !HaveMainExt() then UNDEFINED;
3 d = UInt(Rd); n = UInt(Rn); m = UInt(Rm); setflags = (S == '1');
4 (shift_t, shift_n) = DecodeImmShift(sr_type, imm3:imm2);
5 if d IN {13,15} || n == 13 || m IN {13,15} then UNPREDICTABLE;
```

#### Assembler symbols for all encodings

|                      |                                                                                                                                     |
|----------------------|-------------------------------------------------------------------------------------------------------------------------------------|
| <b>&lt;c&gt;</b>     | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                                                            |
| <b>&lt;q&gt;</b>     | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                                                            |
| <b>&lt;Rd&gt;</b>    | Is the general-purpose destination register, encoded in the "Rd" field.                                                             |
| <b>&lt;Rn&gt;</b>    | Is the first general-purpose source register, encoded in the "Rn" field.                                                            |
| <b>&lt;Rm&gt;</b>    | Is the second general-purpose source register, encoded in the "Rm" field.                                                           |
| <b>&lt;shift&gt;</b> | Is the type of shift to be applied to the second source register, encoded in the "sr_type" field. It can have the following values: |
|                      | LSL when sr_type = 00                                                                                                               |
|                      | LSR when sr_type = 01                                                                                                               |
|                      | ASR when sr_type = 10                                                                                                               |
|                      | ROR when sr_type = 11                                                                                                               |

<amount> Is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR) encoded in the "imm3:imm2" field as <amount> modulo 32.

### Operation for all encodings

```
1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 (shifted, carry) = Shift_C(R[m], shift_t, shift_n, APSR.C);
4 result = R[n] OR NOT(shifted);
5 R[d] = result;
6 if setflags then
7 APSR.N = result[31];
8 APSR.Z = IsZeroBit(result);
9 APSR.C = carry;
10 // APSR.V unchanged
```



### C2.4.104 ORR (immediate)

Logical OR (immediate). Logical OR (immediate) performs a bitwise (inclusive) OR of a register value and an immediate value, and writes the result to the destination register. It can optionally update the condition flags based on the result.

#### T1

Armv8-M Main Extension only

|    |    |    |    |    |    |   |   |   |   |   |   |            |   |      |    |      |    |    |    |    |    |   |   |   |   |   |   |   |   |   |   |
|----|----|----|----|----|----|---|---|---|---|---|---|------------|---|------|----|------|----|----|----|----|----|---|---|---|---|---|---|---|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3          | 2 | 1    | 0  | 15   | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 1  | 1  | 1  | 1  | 0  | i  | 0 | 0 | 0 | 1 | 0 | S | Rn != 1111 | 0 | imm3 | Rd | imm8 |    |    |    |    |    |   |   |   |   |   |   |   |   |   |   |

#### ORR variant

Applies when **S == 0**.

ORR{<c>}{<q>} {<Rd>}, {<Rn>, #<const>

#### ORRS variant

Applies when **S == 1**.

ORRS{<c>}{<q>} {<Rd>}, {<Rn>, #<const>

#### Decode for this encoding

```

1 if Rn == '1111' then SEE "MOV (immediate)";
2 if !HaveMainExt() then UNDEFINED;
3 d = UInt(Rd); n = UInt(Rn); setflags = (S == '1');
4 (imm32, carry) = T32ExpandImm_C(i:imm3:imm8, APSR.C);
5 if d IN {13,15} || n == 13 then UNPREDICTABLE;

```

#### Assembler symbols for all encodings

- <c> See [C1.2.5 Standard assembler syntax fields](#) on page 309.
- <q> See [C1.2.5 Standard assembler syntax fields](#) on page 309.
- <Rd> Is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as <Rn>.
- <Rn> Is the general-purpose source register, encoded in the "Rn" field.
- <const> Is an immediate value derived from the 12-bit immediate that is encoded in the 'i:imm3:imm8' field. See [C1.5 Modified immediate constants](#) on page 324 for the range of values.

#### Operation for all encodings

```

1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 result = R[n] OR imm32;
4 R[d] = result;
5 if setflags then
6 APSR.N = result[31];
7 APSR.Z = IsZeroBit(result);
8 APSR.C = carry;
9 // APSR.V unchanged

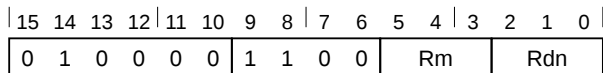
```

### C2.4.105 ORR (register)

Logical OR (register). Logical OR (register) performs a bitwise (inclusive) OR of a register value and an optionally-shifted register value, and writes the result to the destination register. It can optionally update the condition flags based on the result.

#### T1

Armv8-M



#### T1 variant

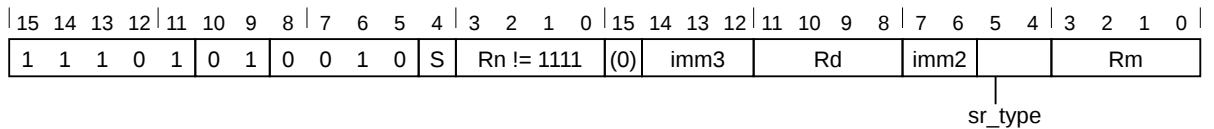
```
ORR<c>{<q>} {<Rdn>, } <Rdn>, <Rm>
 // Inside IT block
ORRS{<q>} {<Rdn>, } <Rdn>, <Rm>
 // Outside IT block
```

#### Decode for this encoding

```
1 d = UInt(Rdn); n = UInt(Rdn); m = UInt(Rm); setflags = !InITBlock();
2 (shift_t, shift_n) = (SRTYPE_LSL, 0);
```

#### T2

Armv8-M Main Extension only



#### ORR, rotate right with extend variant

Applies when **S == 0** && **imm3 == 000** && **imm2 == 00** && **sr\_type == 11**.

```
ORR{<c>}{<q>} {<Rd>, } <Rn>, <Rm>, RRX
```

#### ORR, shift or rotate by value variant

Applies when **S == 0** && **!(imm3 == 000 && imm2 == 00 && sr\_type == 11)**.

```
ORR<c>.W {<Rd>, } <Rn>, <Rm>
 // Inside IT block, and <Rd>, <Rn>, <Rm> can be represented in T1
ORR{<c>}{<q>} {<Rd>, } <Rn>, <Rm> {, <shift> #<amount>}
```

#### ORRS, rotate right with extend variant

Applies when **S == 1** && **imm3 == 000** && **imm2 == 00** && **sr\_type == 11**.

```
ORRS{<c>}{<q>} {<Rd>, } <Rn>, <Rm>, RRX
```

#### ORRS, shift or rotate by value variant

Applies when **S == 1** && **!(imm3 == 000 && imm2 == 00 && sr\_type == 11)**.

```
ORRS.W {<Rd>}, <Rn>, <Rm>
// Outside IT block, and <Rd>, <Rn>, <Rm> can be represented in T1
ORRS{<c>}{<q>} {<Rd>}, <Rn>, <Rm> {, <shift> #<amount>}
```

### Decode for this encoding

```
1 if Rn == '1111' then SEE "MOV (register)";
2 if !HaveMainExt() then UNDEFINED;
3 d = UInt(Rd); n = UInt(Rn); m = UInt(Rm); setflags = (S == '1');
4 (shift_t, shift_n) = DecodeImmShift(sr_type, imm3:imm2);
5 if d IN {13,15} || n == 13 || m IN {13,15} then UNPREDICTABLE;
```

### Assembler symbols for all encodings

<c> See [C1.2.5 Standard assembler syntax fields](#) on page 309.

<q> See [C1.2.5 Standard assembler syntax fields](#) on page 309.

<Rdn> Is the first general-purpose source register and the destination register, encoded in the "Rdn" field.

<Rd> Is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as <Rn>.

<Rn> Is the first general-purpose source register, encoded in the "Rn" field.

<Rm> Is the second general-purpose source register, encoded in the "Rm" field.

<shift> Is the type of shift to be applied to the second source register, encoded in the "sr\_type" field. It can have the following values:

|     |                   |
|-----|-------------------|
| LSL | when sr_type = 00 |
| LSR | when sr_type = 01 |
| ASR | when sr_type = 10 |
| ROR | when sr_type = 11 |

<amount> Is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR) encoded in the "imm3:imm2" field as <amount> modulo 32.

### Operation for all encodings

```
1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 (shifted, carry) = Shift_C(R[m], shift_t, shift_n, APSR.C);
4 result = R[n] OR shifted;
5 R[d] = result;
6 if setflags then
7 APSR.N = result[31];
8 APSR.Z = IsZeroBit(result);
9 APSR.C = carry;
10 // APSR.V unchanged
```



```
2 EncodingSpecificOperations();
3 operand2 = Shift(R[m], shift_t, shift_n, APSR.C); // APSR.C ignored
4 bits(32) result;
5 result[15:0] = if tbform then operand2[15:0] else R[n][15:0];
6 result[31:16] = if tbform then R[n][31:16] else operand2[31:16];
7 R[d] = result;
```

### C2.4.107 PLD (literal)

Preload Data (literal). Preload Data signals the memory system that data memory accesses from a specified address are likely in the near future. The memory system can respond by taking actions that are expected to speed up the memory accesses when they do occur, such as pre-loading the cache line containing the specified address into the data cache.

#### T1

Armv8-M Main Extension only

|    |    |    |    |    |    |   |   |   |   |     |   |   |   |   |   |    |    |    |    |       |    |   |   |   |   |   |   |   |   |   |   |
|----|----|----|----|----|----|---|---|---|---|-----|---|---|---|---|---|----|----|----|----|-------|----|---|---|---|---|---|---|---|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5   | 4 | 3 | 2 | 1 | 0 | 15 | 14 | 13 | 12 | 11    | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 1  | 1  | 1  | 1  | 1  | 0  | 0 | 0 | U | 0 | (0) | 1 | 1 | 1 | 1 | 1 | 1  | 1  | 1  | 1  | imm12 |    |   |   |   |   |   |   |   |   |   |   |

#### T1 variant

```
PLD{<c>}{<q>} <label>
 // Preferred syntax
PLD{<c>}{<q>} [PC, #<+/-><imm>]
 // Alternative syntax
```

#### Decode for this encoding

```
1 if !HaveMainExt() then UNDEFINED;
2 imm32 = ZeroExtend(imm12, 32); add = (U == '1');
```

#### Assembler symbols for all encodings

<c> See [C1.2.5 Standard assembler syntax fields](#) on page 309.

<q> See [C1.2.5 Standard assembler syntax fields](#) on page 309.

<label> The label of the literal data item that is likely to be accessed in the near future. The assembler calculates the required value of the offset from the [Align](#)(PC, 4) value of the instruction to this label. The offset must be in the range -4095 to 4095. If the offset is zero or positive, imm32 is equal to the offset and add == TRUE. If the offset is negative, imm32 is equal to minus the offset and add == FALSE.

+/- Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in the "U" field. It can have the following values:

- when U = 0
- + when U = 1

<imm> Is a 12-bit unsigned immediate byte offset, in the range 0 to 4095, encoded in the "imm12" field.

#### Operation for all encodings

```
1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 address = if add then (Align(PC, 4) + imm32) else (Align(PC, 4) - imm32);
4 Hint_PreloadData(address);
```

### C2.4.108 PLD, PLDW (immediate)

Preload Data (immediate). Preload Data signals the memory system that data memory accesses from a specified address are likely in the near future. The memory system can respond by taking actions that are expected to speed up the memory accesses when they do occur, such as pre-loading the cache line containing the specified address into the data cache.

The PLD instruction signals that the likely memory access is a read, and the PLDW instruction signals that it is a write.

The effect of a PLD or PLDW is IMPLEMENTATION DEFINED.

#### T1

Armv8-M Main Extension only

|    |    |    |    |    |    |   |   |   |   |   |   |            |   |   |   |    |       |    |    |    |    |   |   |   |   |   |   |   |   |   |   |
|----|----|----|----|----|----|---|---|---|---|---|---|------------|---|---|---|----|-------|----|----|----|----|---|---|---|---|---|---|---|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3          | 2 | 1 | 0 | 15 | 14    | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 1  | 1  | 1  | 1  | 1  | 0  | 0 | 0 | 1 | 0 | W | 1 | Rn != 1111 | 1 | 1 | 1 | 1  | imm12 |    |    |    |    |   |   |   |   |   |   |   |   |   |   |

#### Preload read variant

Applies when  $\bar{W} == 0$ .

PLD{<c>}{<q>} [<Rn> {, #<+><imm>}]

#### Preload write variant

Applies when  $\bar{W} == 1$ .

PLDW{<c>}{<q>} [<Rn> {, #<+><imm>}]

#### Decode for this encoding

```

1 if Rn == '1111' then SEE "PLD (literal)";
2 if !HaveMainExt() then UNDEFINED;
3 n = UInt(Rn); imm32 = ZeroExtend(imm12, 32); add = TRUE; is_pldw = (W == '1');
```

#### T2

Armv8-M Main Extension only

|    |    |    |    |    |    |   |   |   |   |   |   |            |   |   |   |    |    |    |    |    |      |   |   |   |   |   |   |   |   |   |   |
|----|----|----|----|----|----|---|---|---|---|---|---|------------|---|---|---|----|----|----|----|----|------|---|---|---|---|---|---|---|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3          | 2 | 1 | 0 | 15 | 14 | 13 | 12 | 11 | 10   | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 1  | 1  | 1  | 1  | 1  | 0  | 0 | 0 | 0 | 0 | W | 1 | Rn != 1111 | 1 | 1 | 1 | 1  | 1  | 1  | 0  | 0  | imm8 |   |   |   |   |   |   |   |   |   |   |

#### Preload read variant

Applies when  $\bar{W} == 0$ .

PLD{<c>}{<q>} [<Rn> {, #<-><imm>}]

#### Preload write variant

Applies when  $\bar{W} == 1$ .

PLDW{<c>}{<q>} [<Rn> {, #<-><imm>}]

#### Decode for this encoding

```

1 if Rn == '1111' then SEE "PLD (literal)";
2 if !HaveMainExt() then UNDEFINED;
3 n = UInt(Rn); imm32 = ZeroExtend(imm8, 32); add = FALSE; is_pldw = (W == '1');
```

### Assembler symbols for all encodings

|       |                                                                                                                                                                                                                                                                                                   |
|-------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <c>   | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                                                                                                                                                                                                                          |
| <q>   | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                                                                                                                                                                                                                          |
| <Rn>  | Is the general-purpose base register, encoded in the "Rn" field. If the PC is used, see <a href="#">C2.4.107 PLD (literal)</a> on page 554.                                                                                                                                                       |
| +     | Specifies the offset is added to the base register.                                                                                                                                                                                                                                               |
| <imm> | For encoding T1: is an optional 12-bit unsigned immediate byte offset, in the range 0 to 4095, defaulting to 0 and encoded in the "imm12" field.<br>For encoding T2: is an optional 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 and encoded in the "imm8" field. |

### Operation for all encodings

```
1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 address = if add then (R[n] + imm32) else (R[n] - imm32);
4 if is_pldw then
5 Hint_PreloadDataForWrite(address);
6 else
7 Hint_PreloadData(address);
```



### C2.4.109 PLD, PLDW (register)

Preload Data (register). Preload Data is a memory hint instruction that can signal the memory system that data memory accesses from a specified address are likely in the near future. The memory system can respond by taking actions that are expected to speed up the memory accesses when they do occur, such as pre-loading the cache line containing the specified address into the data cache.

The PLD instruction signals that the likely memory access is a read, and the PLDW instruction signals that it is a write.

The effect of a PLD or PLDW is IMPLEMENTATION DEFINED.

#### T1

Armv8-M Main Extension only

|    |    |    |    |    |    |   |   |   |   |   |   |            |   |   |   |    |    |    |    |    |    |   |   |   |   |   |   |   |   |   |   |
|----|----|----|----|----|----|---|---|---|---|---|---|------------|---|---|---|----|----|----|----|----|----|---|---|---|---|---|---|---|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3          | 2 | 1 | 0 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 1  | 1  | 1  | 1  | 1  | 0  | 0 | 0 | 0 | 0 | W | 1 | Rn != 1111 | 1 | 1 | 1 | 1  | 0  | 0  | 0  | 0  | 0  | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

#### Preload read variant

Applies when **W** == 0.

PLD{<c>}{<q>} [<Rn>, {+}<Rm> {, LSL #<amount>}]

#### Preload write variant

Applies when **W** == 1.

PLDW{<c>}{<q>} [<Rn>, {+}<Rm> {, LSL #<amount>}]

#### Decode for this encoding

```

1 if Rn == '1111' then SEE "PLD (literal)";
2 if !HaveMainExt() then UNDEFINED;
3 n = UInt(Rn); m = UInt(Rm); add = TRUE; is_pldw = (W == '1');
4 (shift_t, shift_n) = (SRTYPE_LSL, UInt(shift));
5 if m IN {13,15} then UNPREDICTABLE;

```

#### Assembler symbols for all encodings

|          |                                                                                         |
|----------|-----------------------------------------------------------------------------------------|
| <c>      | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                |
| <q>      | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                |
| <Rn>     | Is the general-purpose base register, encoded in the "Rn" field.                        |
| +        | Specifies the index register that is added to the base register.                        |
| <Rm>     | Is the general purpose index register, encoded in the "Rm" field.                       |
| <amount> | Is the shift amount, in the range 0-3, defaulting to 0 and encoded in the "imm2" field. |

#### Operation for all encodings

```

1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 offset = Shift(R[m], shift_t, shift_n, APSR.C);
4 address = if add then (R[n] + offset) else (R[n] - offset);
5 if is_pldw then
6 Hint_PreloadDataForWrite(address);
7 else
8 Hint_PreloadData(address);

```

### C2.4.110 PLI (immediate, literal)

Preload Instruction (immediate, literal). Preload Instruction is a memory hint instruction that can signal the memory system that instruction memory accesses from a specified address are likely in the near future. The memory system can respond by taking actions that are expected to speed up the memory accesses when they do occur, such as pre-loading the cache line containing the specified address into the instruction cache.

#### T1

Armv8-M Main Extension only

|    |    |    |    |    |    |   |   |   |   |   |   |            |   |   |   |    |       |    |    |    |    |   |   |   |   |   |   |   |   |   |   |
|----|----|----|----|----|----|---|---|---|---|---|---|------------|---|---|---|----|-------|----|----|----|----|---|---|---|---|---|---|---|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3          | 2 | 1 | 0 | 15 | 14    | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 1  | 1  | 1  | 1  | 1  | 0  | 0 | 1 | 1 | 0 | 0 | 1 | Rn != 1111 | 1 | 1 | 1 | 1  | imm12 |    |    |    |    |   |   |   |   |   |   |   |   |   |   |

#### T1 variant

PLI{<c>}{<q>} [<Rn> {, #+}<imm>]

#### Decode for this encoding

```
1 if Rn == '1111' then SEE "encoding T3";
2 if !HaveMainExt() then UNDEFINED;
3 n = UInt(Rn); imm32 = ZeroExtend(imm12, 32); add = TRUE;
```

#### T2

Armv8-M Main Extension only

|    |    |    |    |    |    |   |   |   |   |   |   |            |   |   |   |    |    |    |    |    |      |   |   |   |   |   |   |   |   |   |   |
|----|----|----|----|----|----|---|---|---|---|---|---|------------|---|---|---|----|----|----|----|----|------|---|---|---|---|---|---|---|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3          | 2 | 1 | 0 | 15 | 14 | 13 | 12 | 11 | 10   | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 1  | 1  | 1  | 1  | 1  | 0  | 0 | 1 | 0 | 0 | 0 | 1 | Rn != 1111 | 1 | 1 | 1 | 1  | 1  | 1  | 0  | 0  | imm8 |   |   |   |   |   |   |   |   |   |   |

#### T2 variant

PLI{<c>}{<q>} [<Rn> {, #-}<imm>]

#### Decode for this encoding

```
1 if Rn == '1111' then SEE "encoding T3";
2 if !HaveMainExt() then UNDEFINED;
3 n = UInt(Rn); imm32 = ZeroExtend(imm8, 32); add = FALSE;
```

#### T3

Armv8-M Main Extension only

|    |    |    |    |    |    |   |   |   |   |   |   |   |   |   |   |    |    |    |    |       |    |   |   |   |   |   |   |   |   |   |   |
|----|----|----|----|----|----|---|---|---|---|---|---|---|---|---|---|----|----|----|----|-------|----|---|---|---|---|---|---|---|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | 15 | 14 | 13 | 12 | 11    | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 1  | 1  | 1  | 1  | 1  | 0  | 0 | 1 | U | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1  | 1  | 1  | 1  | imm12 |    |   |   |   |   |   |   |   |   |   |   |

#### T3 variant

```
PLI{<c>}{<q>} <label>
 // Preferred syntax
PLI{<c>}{<q>} [PC, #+/-}<imm>]
 // Alternative syntax
```

#### Decode for this encoding

```
1 if !HaveMainExt() then UNDEFINED;
2 n = 15; imm32 = ZeroExtend(imm12, 32); add = (U == '1');
```

## Assembler symbols for all encodings

|         |                                                                                                                                                                                                                                                                                                                                                                                                                                         |
|---------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <c>     | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                                                                                                                                                                                                                                                                                                                                                                |
| <q>     | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                                                                                                                                                                                                                                                                                                                                                                |
| <label> | The label of the instruction that is likely to be accessed in the near future. The assembler calculates the required value of the offset from the <a href="#">Align</a> (PC, 4) value of the instruction to this label. The offset must be in the range -4095 to 4095. If the offset is zero or positive, imm32 is equal to the offset and add == TRUE. If the offset is negative, imm32 is equal to minus the offset and add == FALSE. |
| <Rn>    | Is the general-purpose base register, encoded in the "Rn" field.                                                                                                                                                                                                                                                                                                                                                                        |
| +       | Specifies the offset is added to the base register.                                                                                                                                                                                                                                                                                                                                                                                     |
| +/-     | Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in the "U" field. It can have the following values: <ul style="list-style-type: none"><li>- when U = 0</li><li>+ when U = 1</li></ul>                                                                                                                                                                                     |
| <imm>   | For encoding T1: is an optional 12-bit unsigned immediate byte offset, in the range 0 to 4095, defaulting to 0 and encoded in the "imm12" field.<br>For encoding T2: is an optional 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 and encoded in the "imm8" field.<br>For encoding T3: is a 12-bit unsigned immediate byte offset, in the range 0 to 4095, encoded in the "imm12" field.                 |

## Operation for all encodings

```
1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 base = if n == 15 then Align(PC,4) else R[n];
4 address = if add then (base + imm32) else (base - imm32);
5 Hint_PreloadInstr(address);
```

### C2.4.111 PLI (register)

Preload Instruction (register). Preload Instruction is a memory hint instruction that can signal the memory system that instruction memory accesses from a specified address are likely in the near future. The memory system can respond by taking actions that are expected to speed up the memory accesses when they do occur, such as pre-loading the cache line containing the specified address into the instruction cache.

#### T1

Armv8-M Main Extension only

|    |    |    |    |    |    |   |   |   |   |   |   |            |   |   |   |    |    |    |    |    |    |   |   |   |      |    |   |   |   |   |   |
|----|----|----|----|----|----|---|---|---|---|---|---|------------|---|---|---|----|----|----|----|----|----|---|---|---|------|----|---|---|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3          | 2 | 1 | 0 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6    | 5  | 4 | 3 | 2 | 1 | 0 |
| 1  | 1  | 1  | 1  | 1  | 0  | 0 | 1 | 0 | 0 | 0 | 1 | Rn != 1111 | 1 | 1 | 1 | 1  | 0  | 0  | 0  | 0  | 0  | 0 | 0 | 0 | imm2 | Rm |   |   |   |   |   |

#### T1 variant

PLI{<c>}{<q>} [<Rn>, {+}<Rm> {, LSL #<amount>}]

#### Decode for this encoding

```

1 if Rn == '1111' then SEE "PLI (immediate, literal)";
2 if !HaveMainExt() then UNDEFINED;
3 n = UInt(Rn); m = UInt(Rm); add = TRUE;
4 (shift_t, shift_n) = (SRTYPE_LSL, UInt(imm2));
5 if m IN {13,15} then UNPREDICTABLE;

```

#### Assembler symbols for all encodings

- <c> See [C1.2.5 Standard assembler syntax fields](#) on page 309.
- <q> See [C1.2.5 Standard assembler syntax fields](#) on page 309.
- <Rn> Is the general-purpose base register, encoded in the "Rn" field.
- + Specifies the index register is added to the base register.
- <Rm> Is the general-purpose index register, encoded in the "Rm" field.
- <amount> Is the shift amount, in the range 0 to 3, defaulting to 0 and encoded in the "imm2" field.

#### Operation for all encodings

```

1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 offset = Shift(R[m], shift_t, shift_n, APSR.C);
4 address = if add then (R[n] + offset) else (R[n] - offset);
5 Hint_PreloadInstr(address);

```



`<registers>` For encoding T2: is a list of two or more registers to be loaded, separated by commas and surrounded by { and }. The lowest-numbered register is loaded from the lowest memory address, through to the highest-numbered register from the highest memory address. The registers in the list must be in the range R0-R12, encoded in the "register\_list" field, and can optionally contain one of the LR or the PC. If the LR is in the list, the "M" field is set to 1, otherwise it defaults to 0. If the PC is in the list, the "P" field is set to 1, otherwise it defaults to 0. The PC can be in the list. If it is, the instruction branches to the address loaded to the PC, and: If the PC is in the list:

- The LR must not be in the list.
- The instruction must be either outside any IT block, or the last instruction in an IT block.

For encoding T3: is a list of one or more registers to be loaded, separated by commas and surrounded by { and }. The registers in the list must be in the range R0-R7, encoded in the "register\_list" field, and can optionally include the PC. If the PC is in the list, the "P" field is set to 1, otherwise this field defaults to 0. If the PC is in the list, the instruction must be either outside any IT block, or the last instruction in an IT block.

### Operation for all encodings

The description of [LDM](#), [LDMIA](#), [LDMFD](#) gives the operational pseudocode for this instruction.

### C2.4.113 POP (single register)

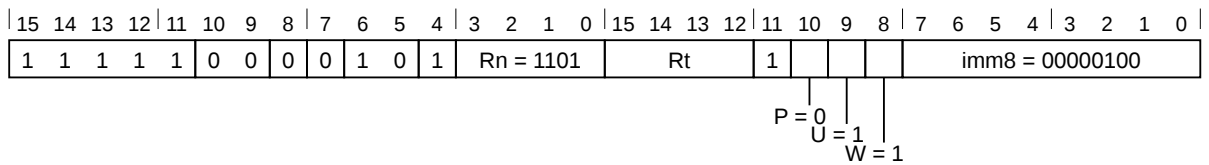
Pop single register from stack. Pop single register from stack loads a single general-purpose register from the stack, loading from the address in SP, and updates SP to point above the loaded data.

This instruction is an alias of the [LDR \(immediate\)](#) instruction. This means that:

- The encodings in this description are named to match the encodings of [LDR \(immediate\)](#).
- The description of [LDR \(immediate\)](#) gives the operational pseudocode for this instruction.

#### T4

*Armv8-M Main Extension only*



#### Post-indexed variant

POP{<c>}{<q>} <register>

#### is equivalent to

LDR{<c>}{<q>} <Rt>, [SP], #4

and is always the preferred disassembly.

#### Assembler symbols for all encodings

- <c> See [C1.2.5 Standard assembler syntax fields](#) on page 309.
- <q> See [C1.2.5 Standard assembler syntax fields](#) on page 309.
- <register> Is the general-purpose register <Rt> to be loaded surrounded by { and }.
- <Rt> Is the general-purpose register to be transferred, encoded in the "Rt" field. The PC can be used, provided the instruction is either outside an IT block or the last instruction of an IT block. If the PC is used, the instruction branches to the address (data) loaded to the PC.

#### Operation for all encodings

The description of [LDR \(immediate\)](#) gives the operational pseudocode for this instruction.

### C2.4.114 PSSBB

Physical Speculative Store Bypass Barrier. Physical Speculative Store Bypass Barrier is a memory barrier which prevents speculative loads from bypassing earlier stores to the same physical address.

The semantics of the Physical Speculative Store Bypass Barrier are:

- When a load to a location appears in program order after the PSSBB, then the load does not speculatively read an entry earlier in the coherence order for that location than the entry generated by the latest store satisfying all of the following conditions:
  - The store is to the same location as the load.
  - The store appears in program order before the PSSBB.
- When a load to a location appears in program order before the PSSBB, then the load does not speculatively read data from any store satisfying all of the following conditions:
  - The store is to the same location as the load.
  - The store appears in program order after the PSSBB.

#### T1

Armv8-M Main Extension only

|  |    |    |    |    |  |    |    |   |   |  |   |   |   |   |  |     |     |     |     |  |    |    |    |     |   |    |     |     |     |     |   |   |   |   |   |   |   |   |   |   |  |
|--|----|----|----|----|--|----|----|---|---|--|---|---|---|---|--|-----|-----|-----|-----|--|----|----|----|-----|---|----|-----|-----|-----|-----|---|---|---|---|---|---|---|---|---|---|--|
|  | 15 | 14 | 13 | 12 |  | 11 | 10 | 9 | 8 |  | 7 | 6 | 5 | 4 |  | 3   | 2   | 1   | 0   |  | 15 | 14 | 13 | 12  |   | 11 | 10  | 9   | 8   |     | 7 | 6 | 5 | 4 |   | 3 | 2 | 1 | 0 |   |  |
|  | 1  | 1  | 1  | 1  |  | 0  | 0  | 1 | 1 |  | 1 | 0 | 1 | 1 |  | (1) | (1) | (1) | (1) |  | 1  | 0  |    | (0) | 0 |    | (1) | (1) | (1) | (1) |   | 0 | 1 | 0 | 0 |   | 0 | 1 | 0 | 0 |  |

#### T1 variant

PSSBB { <q> }

#### Decode for this encoding

```
1 if !HaveMainExt() then UNDEFINED;
2 if InITBlock() then UNPREDICTABLE;
```

#### Assembler symbols for all encodings

<q> See [C1.2.5 Standard assembler syntax fields](#) on page 309.

#### Operation for all encodings

```
1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 SpeculativeSynchronizationBarrier();
```



### C2.4.115 PUSH (multiple registers)

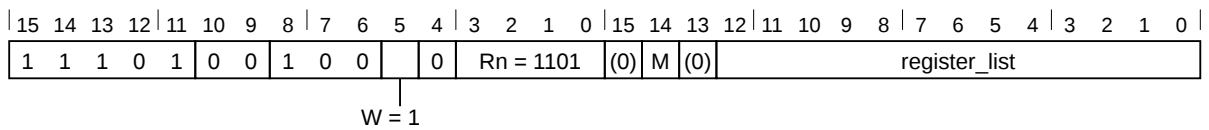
Push multiple registers to stack. Push multiple registers to stack stores multiple general-purpose registers to the stack, storing to consecutive memory locations ending below the address in SP, and updates SP to point to the start of the stored data.

This instruction is an alias of the [STMDB](#), [STMFd](#) instruction. This means that:

- The encodings in this description are named to match the encodings of [STMDB](#), [STMFd](#).
- The description of [STMDB](#), [STMFd](#) gives the operational pseudocode for this instruction.

#### T1

*Armv8-M Main Extension only*



#### T1 variant

PUSH{<c>}{<q>} <registers>

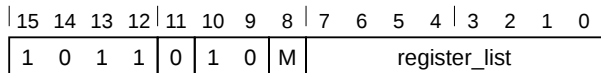
**is equivalent to**

STMDB{<c>}{<q>} SP!, <registers>

**and is the preferred disassembly when `BitCount (register_list) > 1`.**

#### T2

*Armv8-M*



#### T2 variant

PUSH{<c>}{<q>} <registers>

**is equivalent to**

STMDB{<c>}{<q>} SP!, <registers>

**and is always the preferred disassembly.**

#### Assembler symbols for all encodings

- <c> See [C1.2.5 Standard assembler syntax fields](#) on page 309.
- <q> See [C1.2.5 Standard assembler syntax fields](#) on page 309.
- <registers> For encoding T1: is a list of one or more registers to be stored, separated by commas and surrounded by { and }. The lowest-numbered register is stored to the lowest memory address, through to the highest-numbered register to the highest memory address. The registers in the list must be in the range R0-R12, encoded in the "register\_list" field, and can optionally contain the LR. If the LR is in the list, the "M" field is set to 1, otherwise it defaults to 0.  
 For encoding T2: is a list of one or more registers to be stored, separated by commas and surrounded by { and }. The registers in the list must be in the range R0-R7, encoded in the "register\_list" field, and can optionally include the LR. If the LR is in the list, the "M" field is set to 1, otherwise this field defaults to 0.

### **Operation for all encodings**

The description of [STMDB](#), [STMFD](#) gives the operational pseudocode for this instruction.

### C2.4.116 PUSH (single register)

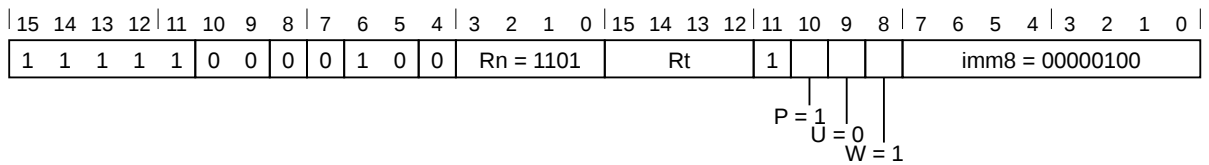
Push single register to stack. Push single register to stack stores a single general-purpose register to the stack, storing to the 32-bit word below the address in SP, and updates SP to point to the start of the stored data.

This instruction is an alias of the [STR \(immediate\)](#) instruction. This means that:

- The encodings in this description are named to match the encodings of [STR \(immediate\)](#).
- The description of [STR \(immediate\)](#) gives the operational pseudocode for this instruction.

#### T4

*Armv8-M Main Extension only*



#### Pre-indexed variant

```
PUSH{<c>}{<q>} <register>
// Standard syntax
```

#### is equivalent to

```
STR{<c>}{<q>} <Rt>, [SP, #-4]!
```

and is always the preferred disassembly.

#### Assembler symbols for all encodings

- <c> See [C1.2.5 Standard assembler syntax fields](#) on page 309.
- <q> See [C1.2.5 Standard assembler syntax fields](#) on page 309.
- <register> Is the general-purpose register <Rt> to be stored surrounded by { and }.
- <Rt> Is the general-purpose register to be transferred, encoded in the "Rt" field.

#### Operation for all encodings

The description of [STR \(immediate\)](#) gives the operational pseudocode for this instruction.

### C2.4.117 QADD

Saturating Add. Saturating Add adds two register values, saturates the result to the 32-bit signed integer range  $-2^{31}$  to  $2^{31}-1$ , and writes the result to the destination register. If saturation occurs, it sets the Q flag in the APSR.

#### T1

*Armv8-M DSP Extension only*

|    |    |    |    |    |    |   |   |   |   |   |   |    |   |   |   |    |    |    |    |    |    |   |   |   |   |   |   |    |   |   |   |
|----|----|----|----|----|----|---|---|---|---|---|---|----|---|---|---|----|----|----|----|----|----|---|---|---|---|---|---|----|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3  | 2 | 1 | 0 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3  | 2 | 1 | 0 |
| 1  | 1  | 1  | 1  | 1  | 0  | 1 | 0 | 1 | 0 | 0 | 0 | Rn |   |   |   | 1  | 1  | 1  | 1  | Rd |    |   |   | 1 | 0 | 0 | 0 | Rm |   |   |   |

#### T1 variant

QADD{<c>}{<q>}{<Rd>}, {<Rm>}, <Rn>

#### Decode for this encoding

```

1 if !HaveDSPExt() then UNDEFINED;
2 d = UInt(Rd); n = UInt(Rn); m = UInt(Rm);
3 if d IN {13,15} || n IN {13,15} || m IN {13,15} then UNPREDICTABLE;

```

#### Assembler symbols for all encodings

<c> See [C1.2.5 Standard assembler syntax fields on page 309](#).  
 <q> See [C1.2.5 Standard assembler syntax fields on page 309](#).  
 <Rd> Is the general-purpose destination register, encoded in the "Rd" field.  
 <Rm> Is the first general-purpose source register, encoded in the "Rm" field.  
 <Rn> Is the second general-purpose source register, encoded in the "Rn" field.

#### Operation for all encodings

```

1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 (R[d], sat) = SignedSatQ(SInt(R[m]) + SInt(R[n]), 32);
4 if sat then
5 APSR.Q = '1';

```

### C2.4.118 QADD16

Saturating Add 16. Saturating Add 16 performs two 16-bit integer additions, saturates the results to the 16-bit signed integer range  $-2^{15}$  to  $2^{15}-1$ , and writes the results to the destination register.

#### T1

Armv8-M DSP Extension only

|    |    |    |    |    |    |   |   |   |   |   |   |    |   |   |   |    |    |    |    |    |    |   |   |   |   |   |   |    |   |   |   |
|----|----|----|----|----|----|---|---|---|---|---|---|----|---|---|---|----|----|----|----|----|----|---|---|---|---|---|---|----|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3  | 2 | 1 | 0 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3  | 2 | 1 | 0 |
| 1  | 1  | 1  | 1  | 1  | 0  | 1 | 0 | 1 | 0 | 0 | 1 | Rn |   |   |   | 1  | 1  | 1  | 1  | Rd |    |   |   | 0 | 0 | 0 | 1 | Rm |   |   |   |

#### T1 variant

QADD16{<c>}{<q>} {<Rd>}, {<Rn>}, <Rm>

#### Decode for this encoding

```

1 if !HaveDSPExt() then UNDEFINED;
2 d = UInt(Rd); n = UInt(Rn); m = UInt(Rm);
3 if d IN {13,15} || n IN {13,15} || m IN {13,15} then UNPREDICTABLE;

```

#### Assembler symbols for all encodings

<c> See [C1.2.5 Standard assembler syntax fields](#) on page 309.  
 <q> See [C1.2.5 Standard assembler syntax fields](#) on page 309.  
 <Rd> Is the general-purpose destination register, encoded in the "Rd" field.  
 <Rn> Is the first general-purpose source register, encoded in the "Rn" field.  
 <Rm> Is the second general-purpose source register, encoded in the "Rm" field.

#### Operation for all encodings

```

1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 sum1 = SInt(R[n][15:0]) + SInt(R[m][15:0]);
4 sum2 = SInt(R[n][31:16]) + SInt(R[m][31:16]);
5 bits(32) result;
6 result[15:0] = SignedSat(sum1, 16);
7 result[31:16] = SignedSat(sum2, 16);
8 R[d] = result;

```

### C2.4.119 QADD8

Saturating Add 8. Saturating Add 8 performs four 8-bit integer additions, saturates the results to the 8-bit signed integer range  $-2^7$  to  $2^7-1$ , and writes the results to the destination register.

#### T1

Armv8-M DSP Extension only

|    |    |    |    |    |    |   |   |   |   |   |   |    |   |   |   |    |    |    |    |    |    |   |   |   |   |   |   |    |   |   |   |
|----|----|----|----|----|----|---|---|---|---|---|---|----|---|---|---|----|----|----|----|----|----|---|---|---|---|---|---|----|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3  | 2 | 1 | 0 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3  | 2 | 1 | 0 |
| 1  | 1  | 1  | 1  | 1  | 0  | 1 | 0 | 1 | 0 | 0 | 0 | Rn |   |   |   | 1  | 1  | 1  | 1  | Rd |    |   |   | 0 | 0 | 0 | 1 | Rm |   |   |   |

#### T1 variant

QADD8 {<c>} {<q>} {<Rd>}, {<Rn>}, {<Rm>}

#### Decode for this encoding

```

1 if !HaveDSPExt() then UNDEFINED;
2 d = UInt(Rd); n = UInt(Rn); m = UInt(Rm);
3 if d IN {13,15} || n IN {13,15} || m IN {13,15} then UNPREDICTABLE;

```

#### Assembler symbols for all encodings

<c> See [C1.2.5 Standard assembler syntax fields](#) on page 309.  
 <q> See [C1.2.5 Standard assembler syntax fields](#) on page 309.  
 <Rd> Is the general-purpose destination register, encoded in the "Rd" field.  
 <Rn> Is the first general-purpose source register, encoded in the "Rn" field.  
 <Rm> Is the second general-purpose source register, encoded in the "Rm" field.

#### Operation for all encodings

```

1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 sum1 = SInt(R[n][7:0]) + SInt(R[m][7:0]);
4 sum2 = SInt(R[n][15:8]) + SInt(R[m][15:8]);
5 sum3 = SInt(R[n][23:16]) + SInt(R[m][23:16]);
6 sum4 = SInt(R[n][31:24]) + SInt(R[m][31:24]);
7 bits(32) result;
8 result[7:0] = SignedSat(sum1, 8);
9 result[15:8] = SignedSat(sum2, 8);
10 result[23:16] = SignedSat(sum3, 8);
11 result[31:24] = SignedSat(sum4, 8);
12 R[d] = result;

```

## C2.4.120 QASX

Saturating Add and Subtract with Exchange. Saturating Add and Subtract with Exchange exchanges the two halfwords of the second operand, performs one 16-bit integer addition and one 16-bit subtraction, saturates the results to the 16-bit signed integer range  $-2^{15}$  to  $2^{15}-1$ , and writes the results to the destination register.

### T1

Armv8-M DSP Extension only

|    |    |    |    |    |    |   |   |   |   |   |   |    |   |   |   |    |    |    |    |    |    |   |   |   |   |   |   |    |   |   |   |
|----|----|----|----|----|----|---|---|---|---|---|---|----|---|---|---|----|----|----|----|----|----|---|---|---|---|---|---|----|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3  | 2 | 1 | 0 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3  | 2 | 1 | 0 |
| 1  | 1  | 1  | 1  | 1  | 0  | 1 | 0 | 1 | 0 | 1 | 0 | Rn |   |   |   | 1  | 1  | 1  | 1  | Rd |    |   |   | 0 | 0 | 0 | 1 | Rm |   |   |   |

### T1 variant

QASX{<c>}{<q>} {<Rd>}, {<Rn>}, <Rm>

### Decode for this encoding

```

1 if !HaveDSPExt() then UNDEFINED;
2 d = UInt(Rd); n = UInt(Rn); m = UInt(Rm);
3 if d IN {13,15} || n IN {13,15} || m IN {13,15} then UNPREDICTABLE;

```

### Assembler symbols for all encodings

<c> See [C1.2.5 Standard assembler syntax fields](#) on page 309.  
 <q> See [C1.2.5 Standard assembler syntax fields](#) on page 309.  
 <Rd> Is the general-purpose destination register, encoded in the "Rd" field.  
 <Rn> Is the first general-purpose source register, encoded in the "Rn" field.  
 <Rm> Is the second general-purpose source register, encoded in the "Rm" field.

### Operation for all encodings

```

1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 diff = SInt(R[n][15:0]) - SInt(R[m][31:16]);
4 sum = SInt(R[n][31:16]) + SInt(R[m][15:0]);
5 bits(32) result;
6 result[15:0] = SignedSat(diff, 16);
7 result[31:16] = SignedSat(sum, 16);
8 R[d] = result;

```

## C2.4.121 QDADD

Saturating Double and Add. Saturating Double and Add adds a doubled register value to another register value, and writes the result to the destination register. Both the doubling and the addition have their results saturated to the 32-bit signed integer range  $-2^{31}$  to  $2^{31}-1$ . If saturation occurs in either operation, it sets the Q flag in the APSR.

### T1

Armv8-M DSP Extension only

|    |    |    |    |    |    |   |   |   |   |   |   |    |   |   |   |    |    |    |    |    |    |   |   |   |   |   |   |    |   |   |   |
|----|----|----|----|----|----|---|---|---|---|---|---|----|---|---|---|----|----|----|----|----|----|---|---|---|---|---|---|----|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3  | 2 | 1 | 0 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3  | 2 | 1 | 0 |
| 1  | 1  | 1  | 1  | 1  | 0  | 1 | 0 | 1 | 0 | 0 | 0 | Rn |   |   |   | 1  | 1  | 1  | 1  | Rd |    |   |   | 1 | 0 | 0 | 1 | Rm |   |   |   |

### T1 variant

QDADD {<c>} {<q>} {<Rd>}, {<Rm>}, <Rn>

### Decode for this encoding

```

1 if !HaveDSPExt() then UNDEFINED;
2 d = UInt(Rd); n = UInt(Rn); m = UInt(Rm);
3 if d IN {13,15} || n IN {13,15} || m IN {13,15} then UNPREDICTABLE;

```

### Assembler symbols for all encodings

<c> See [C1.2.5 Standard assembler syntax fields](#) on page 309.  
 <q> See [C1.2.5 Standard assembler syntax fields](#) on page 309.  
 <Rd> Is the general-purpose destination register, encoded in the "Rd" field.  
 <Rm> Is the first general-purpose source register, encoded in the "Rm" field.  
 <Rn> Is the second general-purpose source register, encoded in the "Rn" field.

### Operation for all encodings

```

1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 (doubled, sat1) = SignedSatQ(2 * SInt(R[n]), 32);
4 (R[d], sat2) = SignedSatQ(SInt(R[m]) + SInt(doubled), 32);
5 if sat1 || sat2 then
6 APSR.Q = '1';

```



## C2.4.122 QDSUB

Saturating Double and Subtract. Saturating Double and Subtract subtracts a doubled register value from another register value, and writes the result to the destination register. Both the doubling and the subtraction have their results saturated to the 32-bit signed integer range  $-2^{31}$  to  $2^{31}-1$ . If saturation occurs in either operation, it sets the Q flag in the APSR.

### T1

Armv8-M DSP Extension only

|    |    |    |    |    |    |   |   |   |   |   |   |    |   |   |   |    |    |    |    |    |    |   |   |   |   |   |   |    |   |   |   |
|----|----|----|----|----|----|---|---|---|---|---|---|----|---|---|---|----|----|----|----|----|----|---|---|---|---|---|---|----|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3  | 2 | 1 | 0 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3  | 2 | 1 | 0 |
| 1  | 1  | 1  | 1  | 1  | 0  | 1 | 0 | 1 | 0 | 0 | 0 | Rn |   |   |   | 1  | 1  | 1  | 1  | Rd |    |   |   | 1 | 0 | 1 | 1 | Rm |   |   |   |

### T1 variant

QDSUB{<c>}{<q>} {<Rd>}, {<Rm>}, <Rn>

### Decode for this encoding

```

1 if !HaveDSPExt() then UNDEFINED;
2 d = UInt(Rd); n = UInt(Rn); m = UInt(Rm);
3 if d IN {13,15} || n IN {13,15} || m IN {13,15} then UNPREDICTABLE;

```

### Assembler symbols for all encodings

<c> See [C1.2.5 Standard assembler syntax fields](#) on page 309.  
 <q> See [C1.2.5 Standard assembler syntax fields](#) on page 309.  
 <Rd> Is the general-purpose destination register, encoded in the "Rd" field.  
 <Rm> Is the first general-purpose source register, encoded in the "Rm" field.  
 <Rn> Is the second general-purpose source register, encoded in the "Rn" field.

### Operation for all encodings

```

1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 (doubled, sat1) = SignedSatQ(2 * SInt(R[n]), 32);
4 (R[d], sat2) = SignedSatQ(SInt(R[m]) - SInt(doubled), 32);
5 if sat1 || sat2 then
6 APSR.Q = '1';

```

## C2.4.123 QSAX

Saturating Subtract and Add with Exchange. Saturating Subtract and Add with Exchange exchanges the two halfwords of the second operand, performs one 16-bit integer subtraction and one 16-bit addition, saturates the results to the 16-bit signed integer range  $-2^{15}$  to  $2^{15}-1$ , and writes the results to the destination register.

### T1

Armv8-M DSP Extension only

|    |    |    |    |    |    |   |   |   |   |   |   |    |   |   |   |    |    |    |    |    |    |   |   |   |   |   |   |    |   |   |   |
|----|----|----|----|----|----|---|---|---|---|---|---|----|---|---|---|----|----|----|----|----|----|---|---|---|---|---|---|----|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3  | 2 | 1 | 0 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3  | 2 | 1 | 0 |
| 1  | 1  | 1  | 1  | 1  | 0  | 1 | 0 | 1 | 1 | 1 | 0 | Rn |   |   |   | 1  | 1  | 1  | 1  | Rd |    |   |   | 0 | 0 | 0 | 1 | Rm |   |   |   |

### T1 variant

QSAX{<c>}{<q>} {<Rd>}, {<Rn>}, <Rm>

### Decode for this encoding

```

1 if !HaveDSPExt() then UNDEFINED;
2 d = UInt(Rd); n = UInt(Rn); m = UInt(Rm);
3 if d IN {13,15} || n IN {13,15} || m IN {13,15} then UNPREDICTABLE;

```

### Assembler symbols for all encodings

<c> See [C1.2.5 Standard assembler syntax fields](#) on page 309.  
 <q> See [C1.2.5 Standard assembler syntax fields](#) on page 309.  
 <Rd> Is the general-purpose destination register, encoded in the "Rd" field.  
 <Rn> Is the first general-purpose source register, encoded in the "Rn" field.  
 <Rm> Is the second general-purpose source register, encoded in the "Rm" field.

### Operation for all encodings

```

1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 sum = SInt(R[n][15:0]) + SInt(R[m][31:16]);
4 diff = SInt(R[n][31:16]) - SInt(R[m][15:0]);
5 bits(32) result;
6 result[15:0] = SignedSat(sum, 16);
7 result[31:16] = SignedSat(diff, 16);
8 R[d] = result;

```

## C2.4.124 QSUB

Saturating Subtract. Saturating Subtract subtracts one register value from another register value, saturates the result to the 32-bit signed integer range  $-2^{31}$  to  $2^{31}-1$ , and writes the result to the destination register. If saturation occurs, it sets the Q flag in the APSR.

### T1

Armv8-M DSP Extension only

|    |    |    |    |    |    |   |   |   |   |   |   |    |   |   |   |    |    |    |    |    |    |   |   |   |   |   |   |    |   |   |   |
|----|----|----|----|----|----|---|---|---|---|---|---|----|---|---|---|----|----|----|----|----|----|---|---|---|---|---|---|----|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3  | 2 | 1 | 0 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3  | 2 | 1 | 0 |
| 1  | 1  | 1  | 1  | 1  | 0  | 1 | 0 | 1 | 0 | 0 | 0 | Rn |   |   |   | 1  | 1  | 1  | 1  | Rd |    |   |   | 1 | 0 | 1 | 0 | Rm |   |   |   |

### T1 variant

QSUB{<c>}{<q>} {<Rd>}, {<Rm>}, <Rn>

### Decode for this encoding

```

1 if !HaveDSPExt() then UNDEFINED;
2 d = UInt(Rd); n = UInt(Rn); m = UInt(Rm);
3 if d IN {13,15} || n IN {13,15} || m IN {13,15} then UNPREDICTABLE;

```

### Assembler symbols for all encodings

<c> See [C1.2.5 Standard assembler syntax fields](#) on page 309.  
 <q> See [C1.2.5 Standard assembler syntax fields](#) on page 309.  
 <Rd> Is the general-purpose destination register, encoded in the "Rd" field.  
 <Rm> Is the first general-purpose source register, encoded in the "Rm" field.  
 <Rn> Is the second general-purpose source register, encoded in the "Rn" field.

### Operation for all encodings

```

1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 (R[d], sat) = SignedSatQ(SInt(R[m]) - SInt(R[n]), 32);
4 if sat then
5 APSR.Q = '1';

```

## C2.4.125 QSUB16

Saturating Subtract 16. Saturating Subtract 16 performs two 16-bit integer subtractions, saturates the results to the 16-bit signed integer range  $-2^{15}$  to  $2^{15}-1$ , and writes the results to the destination register.

### T1

Armv8-M DSP Extension only

|    |    |    |    |    |    |   |   |   |   |   |   |    |   |   |   |    |    |    |    |    |    |   |   |   |   |   |   |    |   |   |   |
|----|----|----|----|----|----|---|---|---|---|---|---|----|---|---|---|----|----|----|----|----|----|---|---|---|---|---|---|----|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3  | 2 | 1 | 0 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3  | 2 | 1 | 0 |
| 1  | 1  | 1  | 1  | 1  | 0  | 1 | 0 | 1 | 1 | 0 | 1 | Rn |   |   |   | 1  | 1  | 1  | 1  | Rd |    |   |   | 0 | 0 | 0 | 1 | Rm |   |   |   |

### T1 variant

QSUB16{<c>}{<q>} {<Rd>}, {<Rn>}, <Rm>

### Decode for this encoding

```

1 if !HaveDSPExt() then UNDEFINED;
2 d = UInt(Rd); n = UInt(Rn); m = UInt(Rm);
3 if d IN {13,15} || n IN {13,15} || m IN {13,15} then UNPREDICTABLE;

```

### Assembler symbols for all encodings

<c> See [C1.2.5 Standard assembler syntax fields](#) on page 309.  
 <q> See [C1.2.5 Standard assembler syntax fields](#) on page 309.  
 <Rd> Is the general-purpose destination register, encoded in the "Rd" field.  
 <Rn> Is the first general-purpose source register, encoded in the "Rn" field.  
 <Rm> Is the second general-purpose source register, encoded in the "Rm" field.

### Operation for all encodings

```

1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 diff1 = SInt(R[n][15:0]) - SInt(R[m][15:0]);
4 diff2 = SInt(R[n][31:16]) - SInt(R[m][31:16]);
5 bits(32) result;
6 result[15:0] = SignedSat(diff1, 16);
7 result[31:16] = SignedSat(diff2, 16);
8 R[d] = result;

```

## C2.4.126 QSUB8

Saturating Subtract 8. Saturating Subtract 8 performs four 8-bit integer subtractions, saturates the results to the 8-bit signed integer range  $-2^7$  to  $2^7-1$ , and writes the results to the destination register.

### T1

Armv8-M DSP Extension only

|    |    |    |    |    |    |   |   |   |   |   |   |    |   |   |   |    |    |    |    |    |    |   |   |   |   |   |   |    |   |   |   |
|----|----|----|----|----|----|---|---|---|---|---|---|----|---|---|---|----|----|----|----|----|----|---|---|---|---|---|---|----|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3  | 2 | 1 | 0 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3  | 2 | 1 | 0 |
| 1  | 1  | 1  | 1  | 1  | 0  | 1 | 0 | 1 | 1 | 0 | 0 | Rn |   |   |   | 1  | 1  | 1  | 1  | Rd |    |   |   | 0 | 0 | 0 | 1 | Rm |   |   |   |

### T1 variant

QSUB8 {<c>} {<q>} {<Rd>}, {<Rn>}, {<Rm>}

### Decode for this encoding

```

1 if !HaveDSPExt() then UNDEFINED;
2 d = UInt(Rd); n = UInt(Rn); m = UInt(Rm);
3 if d IN {13,15} || n IN {13,15} || m IN {13,15} then UNPREDICTABLE;

```

### Assembler symbols for all encodings

<c> See [C1.2.5 Standard assembler syntax fields](#) on page 309.  
 <q> See [C1.2.5 Standard assembler syntax fields](#) on page 309.  
 <Rd> Is the general-purpose destination register, encoded in the "Rd" field.  
 <Rn> Is the first general-purpose source register, encoded in the "Rn" field.  
 <Rm> Is the second general-purpose source register, encoded in the "Rm" field.

### Operation for all encodings

```

1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 diff1 = SInt(R[n][7:0]) - SInt(R[m][7:0]);
4 diff2 = SInt(R[n][15:8]) - SInt(R[m][15:8]);
5 diff3 = SInt(R[n][23:16]) - SInt(R[m][23:16]);
6 diff4 = SInt(R[n][31:24]) - SInt(R[m][31:24]);
7 R[d][7:0] = SignedSat(diff1, 8);
8 R[d][15:8] = SignedSat(diff2, 8);
9 R[d][23:16] = SignedSat(diff3, 8);
10 R[d][31:24] = SignedSat(diff4, 8);

```

## C2.4.127 RBIT

Reverse Bits. Reverse Bits reverses the bit order in a 32-bit register.

### T1

Armv8-M Main Extension only

|    |    |    |    |    |    |   |   |   |   |   |   |    |   |   |   |    |    |    |    |    |    |   |   |   |   |   |   |     |   |   |   |
|----|----|----|----|----|----|---|---|---|---|---|---|----|---|---|---|----|----|----|----|----|----|---|---|---|---|---|---|-----|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3  | 2 | 1 | 0 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3   | 2 | 1 | 0 |
| 1  | 1  | 1  | 1  | 1  | 0  | 1 | 0 | 1 | 0 | 0 | 1 | Rm |   |   |   | 1  | 1  | 1  | 1  | Rd |    |   |   | 1 | 0 | 1 | 0 | Rm2 |   |   |   |

### T1 variant

RBIT{<c>} {<q>} <Rd>, <Rm>

### Decode for this encoding

```

1 if !HaveMainExt() then UNDEFINED;
2 if Rm != Rm2 then UNPREDICTABLE;
3 d = UInt(Rd); m = UInt(Rm);
4 if d IN {13,15} || m IN {13,15} then UNPREDICTABLE;
```

### CONSTRAINED UNPREDICTABLE behavior

If  $Rm \neq Rm2$ , then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes as described, with no change to its behavior and no additional side effects.
- The value in the destination register is UNKNOWN.

### Assembler symbols for all encodings

|      |                                                                                                      |
|------|------------------------------------------------------------------------------------------------------|
| <c>  | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                             |
| <q>  | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                             |
| <Rd> | Is the general-purpose destination register, encoded in the "Rd" field.                              |
| <Rm> | Is the general-purpose source register, encoded in the "Rm" field. Its number must be encoded twice. |

### Operation for all encodings

```

1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 bits(32) result;
4 for i = 0 to 31
5 result[31-i] = R[m][i];
6 R[d] = result;
```

## C2.4.128 REV

Byte-Reverse Word. Byte-Reverse Word reverses the byte order in a 32-bit register.

### T1

*Armv8-M*

|    |    |    |    |    |    |   |   |   |   |    |   |   |   |    |   |
|----|----|----|----|----|----|---|---|---|---|----|---|---|---|----|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5  | 4 | 3 | 2 | 1  | 0 |
| 1  | 0  | 1  | 1  | 1  | 0  | 1 | 0 | 0 | 0 | Rm |   |   |   | Rd |   |

### T1 variant

REV{<c>}{<q>} <Rd>, <Rm>

### Decode for this encoding

```
1 d = UInt(Rd); m = UInt(Rm);
```

### T2

*Armv8-M Main Extension only*

|    |    |    |    |    |    |   |   |   |   |   |   |    |   |   |   |    |    |    |    |    |    |   |   |   |   |   |   |   |   |     |   |
|----|----|----|----|----|----|---|---|---|---|---|---|----|---|---|---|----|----|----|----|----|----|---|---|---|---|---|---|---|---|-----|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3  | 2 | 1 | 0 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1   | 0 |
| 1  | 1  | 1  | 1  | 1  | 0  | 1 | 0 | 1 | 0 | 0 | 1 | Rm |   |   |   | 1  | 1  | 1  | 1  | Rd |    |   |   | 1 | 0 | 0 | 0 |   |   | Rm2 |   |

### T2 variant

REV{<c>}.W <Rd>, <Rm>  
 // <Rd>, <Rm> can be represented in T1  
 REV{<c>}{<q>} <Rd>, <Rm>

### Decode for this encoding

```
1 if !HaveMainExt() then UNDEFINED;
2 if Rm != Rm2 then UNPREDICTABLE;
3 d = UInt(Rd); m = UInt(Rm);
4 if d IN {13,15} || m IN {13,15} then UNPREDICTABLE;
```

### CONSTRAINED UNPREDICTABLE behavior

If  $Rm \neq Rm2$ , then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes as described, with no change to its behavior and no additional side effects.
- The value in the destination register is UNKNOWN.

### Assembler symbols for all encodings

|      |                                                                                                                                                                                                              |
|------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <c>  | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                                                                                                                                     |
| <q>  | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                                                                                                                                     |
| <Rd> | Is the general-purpose destination register, encoded in the "Rd" field.                                                                                                                                      |
| <Rm> | For encoding T1: is the general-purpose source register, encoded in the "Rm" field.<br>For encoding T2: is the general-purpose source register, encoded in the "Rm" field. Its number must be encoded twice. |

### Operation for all encodings

```
1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 bits(32) result;
4 result[31:24] = R[m][7:0];
5 result[23:16] = R[m][15:8];
6 result[15:8] = R[m][23:16];
7 result[7:0] = R[m][31:24];
8 R[d] = result;
```

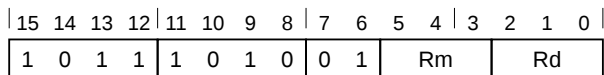


## C2.4.129 REV16

Byte-Reverse Packed Halfword. Byte-Reverse Packed Halfword reverses the byte order in each 16-bit halfword of a 32-bit register.

### T1

Armv8-M



### T1 variant

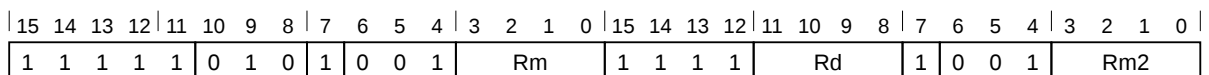
REV16{<c>}{<q>} <Rd>, <Rm>

### Decode for this encoding

```
1 d = UInt(Rd); m = UInt(Rm);
```

### T2

Armv8-M Main Extension only



### T2 variant

```
REV16{<c>}.W <Rd>, <Rm>
// <Rd>, <Rm> can be represented in T1
REV16{<c>}{<q>} <Rd>, <Rm>
```

### Decode for this encoding

```
1 if !HaveMainExt() then UNDEFINED;
2 if Rm != Rm2 then UNPREDICTABLE;
3 d = UInt(Rd); m = UInt(Rm);
4 if d IN {13,15} || m IN {13,15} then UNPREDICTABLE;
```

### CONSTRAINED UNPREDICTABLE behavior

If  $Rm \neq Rm2$ , then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes as described, with no change to its behavior and no additional side effects.
- The value in the destination register is UNKNOWN.

### Assembler symbols for all encodings

<c> See [C1.2.5 Standard assembler syntax fields](#) on page 309.  
 <q> See [C1.2.5 Standard assembler syntax fields](#) on page 309.  
 <Rd> Is the general-purpose destination register, encoded in the "Rd" field.  
 <Rm> For encoding T1: is the general-purpose source register, encoded in the "Rm" field.  
 For encoding T2: is the general-purpose source register, encoded in the "Rm" field. Its number must be encoded twice.

### Operation for all encodings

```
1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 bits(32) result;
4 result[31:24] = R[m][23:16];
5 result[23:16] = R[m][31:24];
6 result[15:8] = R[m][7:0];
7 result[7:0] = R[m][15:8];
8 R[d] = result;
```

## C2.4.130 REVSH

Byte-Reverse Signed Halfword. Byte-Reverse Signed Halfword reverses the byte order in the lower 16-bit halfword of a 32-bit register, and sign extends the result to 32 bits.

### T1

Armv8-M

|    |    |    |    |    |    |   |   |   |   |   |    |   |   |   |    |
|----|----|----|----|----|----|---|---|---|---|---|----|---|---|---|----|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4  | 3 | 2 | 1 | 0  |
| 1  | 0  | 1  | 1  | 1  | 0  | 1 | 0 | 1 | 1 |   | Rm |   |   |   | Rd |

### T1 variant

REVSH{<c>}{<q>} <Rd>, <Rm>

### Decode for this encoding

```
1 d = UInt(Rd); m = UInt(Rm);
```

### T2

Armv8-M Main Extension only

|    |    |    |    |    |    |   |   |   |   |   |   |   |   |    |   |    |    |    |    |    |    |   |   |   |   |   |   |   |   |     |   |
|----|----|----|----|----|----|---|---|---|---|---|---|---|---|----|---|----|----|----|----|----|----|---|---|---|---|---|---|---|---|-----|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1  | 0 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1   | 0 |
| 1  | 1  | 1  | 1  | 1  | 0  | 1 | 0 | 1 | 1 | 0 | 0 | 1 |   | Rm |   | 1  | 1  | 1  | 1  |    | Rd |   | 1 | 0 | 1 | 1 |   |   |   | Rm2 |   |

### T2 variant

REVSH{<c>}.W <Rd>, <Rm>  
 // <Rd>, <Rm> can be represented in T1  
 REVSH{<c>}{<q>} <Rd>, <Rm>

### Decode for this encoding

```
1 if !HaveMainExt() then UNDEFINED;
2 if Rm != Rm2 then UNPREDICTABLE;
3 d = UInt(Rd); m = UInt(Rm);
4 if d IN {13,15} || m IN {13,15} then UNPREDICTABLE;
```

### CONSTRAINED UNPREDICTABLE behavior

If  $Rm \neq Rm2$ , then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes as described, with no change to its behavior and no additional side effects.
- The value in the destination register is UNKNOWN.

### Assembler symbols for all encodings

|      |                                                                                                                                                                                                              |
|------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <c>  | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                                                                                                                                     |
| <q>  | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                                                                                                                                     |
| <Rd> | Is the general-purpose destination register, encoded in the "Rd" field.                                                                                                                                      |
| <Rm> | For encoding T1: is the general-purpose source register, encoded in the "Rm" field.<br>For encoding T2: is the general-purpose source register, encoded in the "Rm" field. Its number must be encoded twice. |

### Operation for all encodings

```
1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 bits(32) result;
4 result[31:8] = SignExtend(R[m][7:0], 24);
5 result[7:0] = R[m][15:8];
6 R[d] = result;
```



### C2.4.132 ROR (register)

Rotate Right (register). Rotate Right (register) rotates a register value by a variable number of bits, inserting the bits that are rotated off the right end into the vacated bit positions on the left, and writes the result to the destination register. The variable number of bits is read from the bottom byte of a register.

This instruction is an alias of the [MOV, MOVS \(register-shifted register\)](#) instruction. This means that:

- The encodings in this description are named to match the encodings of [MOV, MOVS \(register-shifted register\)](#).
- The description of [MOV, MOVS \(register-shifted register\)](#) gives the operational pseudocode for this instruction.

#### T1

*Armv8-M Main Extension only*

|    |    |    |    |    |    |           |   |    |   |   |   |     |   |   |   |
|----|----|----|----|----|----|-----------|---|----|---|---|---|-----|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9         | 8 | 7  | 6 | 5 | 4 | 3   | 2 | 1 | 0 |
| 0  | 1  | 0  | 0  | 0  | 0  | op = 0111 |   | Rs |   |   |   | Rdm |   |   |   |

#### Rotate right variant

```
ROR<c>{<q>} {<Rdm>}, <Rdm>, <Rs>
// Inside IT block
```

#### is equivalent to

```
MOV<c>{<q>} <Rdm>, <Rdm>, ROR <Rs>
```

**and is the preferred disassembly when `InITBlock()`.**

#### T2

*Armv8-M Main Extension only*

|    |    |    |    |    |    |   |   |   |   |   |   |   |    |   |   |    |    |    |    |    |    |    |   |   |   |   |   |   |   |   |    |  |  |  |
|----|----|----|----|----|----|---|---|---|---|---|---|---|----|---|---|----|----|----|----|----|----|----|---|---|---|---|---|---|---|---|----|--|--|--|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2  | 1 | 0 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0  |  |  |  |
| 1  | 1  | 1  | 1  | 1  | 0  | 1 | 0 | 0 |   |   |   |   | Rm |   |   |    | 1  | 1  | 1  | 1  |    | Rd |   |   |   | 0 | 0 | 0 | 0 |   | Rs |  |  |  |

$\begin{matrix} | \\ \text{sr\_type} = 11 \\ \text{S} = 0 \end{matrix}$

#### Non flag setting variant

```
ROR<c>.W {<Rd>}, <Rm>, <Rs>
// Inside IT block, and <Rd>, <Rm>, <sr_type>, <Rs> can be represented in T1
```

#### is equivalent to

```
MOV{<c>}{<q>} <Rd>, <Rm>, ROR <Rs>
```

**and is always the preferred disassembly.**

#### Non flag setting variant

```
ROR{<c>}{<q>} {<Rd>}, <Rm>, <Rs>
```

#### is equivalent to

```
MOV{<c>}{<q>} <Rd>, <Rm>, ROR <Rs>
```

**and is always the preferred disassembly.**

### Assembler symbols for all encodings

|       |                                                                                                                        |
|-------|------------------------------------------------------------------------------------------------------------------------|
| <c>   | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                                               |
| <q>   | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                                               |
| <Rdm> | Is the first general-purpose source register and the destination register, encoded in the "Rdm" field.                 |
| <Rd>  | Is the general-purpose destination register, encoded in the "Rd" field.                                                |
| <Rm>  | Is the first general-purpose source register, encoded in the "Rm" field.                                               |
| <Rs>  | Is the second general-purpose source register holding a rotate amount in its bottom 8 bits, encoded in the "Rs" field. |

### Operation for all encodings

The description of [MOV, MOVS \(register-shifted register\)](#) gives the operational pseudocode for this instruction.

### C2.4.133 RORS (immediate)

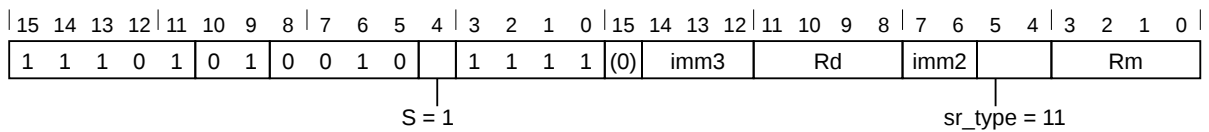
Rotate Right, Setting flags (immediate). Rotate Right, Setting flags (immediate) rotates a register value by a constant number of bits, inserting the bits that are rotated off the right end into the vacated bit positions on the left, writes the result to the destination register, and updates the condition flags based on the result.

This instruction is an alias of the [MOV \(register\)](#) instruction. This means that:

- The encodings in this description are named to match the encodings of [MOV \(register\)](#).
- The description of [MOV \(register\)](#) gives the operational pseudocode for this instruction.

#### T3

*Armv8-M Main Extension only*



#### MOVS, shift or rotate by value variant

Applies when **!(imm3 == 000 && imm2 == 00)**.

RORS {<c>} {<q>} {<Rd>}, {<Rm>}, #<imm>

#### is equivalent to

MOVS {<c>} {<q>} <Rd>, <Rm>, ROR #<imm>

and is always the preferred disassembly.

#### Assembler symbols for all encodings

- <c> See [C1.2.5 Standard assembler syntax fields](#) on page 309.
- <q> See [C1.2.5 Standard assembler syntax fields](#) on page 309.
- <Rd> Is the general-purpose destination register, encoded in the "Rd" field.
- <Rm> Is the general-purpose source register, encoded in the "Rm" field.
- <imm> Is the shift amount, in the range 1 to 31, encoded in the "imm3:imm2" field.

#### Operation for all encodings

The description of [MOV \(register\)](#) gives the operational pseudocode for this instruction.



### C2.4.134 RORS (register)

Rotate Right, Setting flags (register). Rotate Right, Setting flags (register) rotates a register value by a variable number of bits, inserting the bits that are rotated off the right end into the vacated bit positions on the left, writes the result to the destination register, and updates the condition flags based on the result. The variable number of bits is read from the bottom byte of a register.

This instruction is an alias of the [MOV, MOVS \(register-shifted register\)](#) instruction. This means that:

- The encodings in this description are named to match the encodings of [MOV, MOVS \(register-shifted register\)](#).
- The description of [MOV, MOVS \(register-shifted register\)](#) gives the operational pseudocode for this instruction.

#### T1

Armv8-M

|    |    |    |    |    |    |           |   |   |   |    |   |   |     |   |   |
|----|----|----|----|----|----|-----------|---|---|---|----|---|---|-----|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9         | 8 | 7 | 6 | 5  | 4 | 3 | 2   | 1 | 0 |
| 0  | 1  | 0  | 0  | 0  | 0  | op = 0111 |   |   |   | Rs |   |   | Rdm |   |   |

#### Rotate right variant

```
RORS{<q>} {<Rdm>, } <Rdm>, <Rs>
// Outside IT block
```

#### is equivalent to

```
MOVS{<q>} <Rdm>, <Rdm>, ROR <Rs>
```

**and is the preferred disassembly when !InITBlock().**

#### T2

Armv8-M Main Extension only

|    |    |    |    |    |    |   |   |   |   |   |   |    |   |   |   |    |    |    |    |    |    |   |   |   |   |    |   |   |   |   |   |
|----|----|----|----|----|----|---|---|---|---|---|---|----|---|---|---|----|----|----|----|----|----|---|---|---|---|----|---|---|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3  | 2 | 1 | 0 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5  | 4 | 3 | 2 | 1 | 0 |
| 1  | 1  | 1  | 1  | 1  | 0  | 1 | 0 | 0 |   |   |   | Rm |   |   | 1 | 1  | 1  | 1  | Rd |    |    | 0 | 0 | 0 | 0 | Rs |   |   |   |   |   |

$sr\_type = 11$   
 $S = 1$

#### Flag setting variant

```
RORS.W {<Rd>, } <Rm>, <Rs>
// Outside IT block, and <Rd>, <Rm>, <sr_type>, <Rs> can be represented in T1
```

#### is equivalent to

```
MOVS{<c>}{<q>} <Rd>, <Rm>, ROR <Rs>
```

**and is always the preferred disassembly.**

#### Flag setting variant

```
RORS{<c>}{<q>} {<Rd>, } <Rm>, <Rs>
```

#### is equivalent to

```
MOVS{<c>}{<q>} <Rd>, <Rm>, ROR <Rs>
```

**and is always the preferred disassembly.**

### Assembler symbols for all encodings

|       |                                                                                                                        |
|-------|------------------------------------------------------------------------------------------------------------------------|
| <c>   | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                                               |
| <q>   | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                                               |
| <Rdm> | Is the first general-purpose source register and the destination register, encoded in the "Rdm" field.                 |
| <Rd>  | Is the general-purpose destination register, encoded in the "Rd" field.                                                |
| <Rm>  | Is the first general-purpose source register, encoded in the "Rm" field.                                               |
| <Rs>  | Is the second general-purpose source register holding a rotate amount in its bottom 8 bits, encoded in the "Rs" field. |

### Operation for all encodings

The description of [MOV, MOVS \(register-shifted register\)](#) gives the operational pseudocode for this instruction.

### C2.4.135 RRX

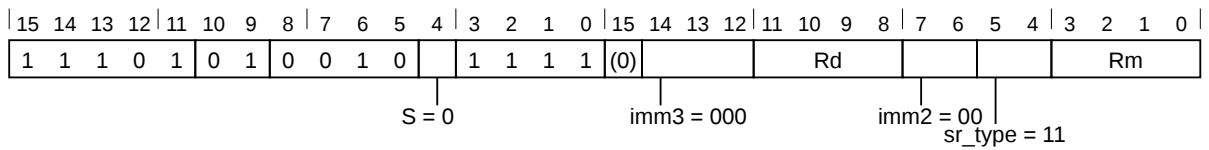
Rotate Right with Extend. Rotate Right with Extend shifts a register value right by one bit, shifting the Carry flag into bit[31], and writes the result to the destination register.

This instruction is an alias of the [MOV \(register\)](#) instruction. This means that:

- The encodings in this description are named to match the encodings of [MOV \(register\)](#).
- The description of [MOV \(register\)](#) gives the operational pseudocode for this instruction.

#### T3

*Armv8-M Main Extension only*



#### MOV, rotate right with extend variant

RRX{<c>} {<q>} {<Rd>}, <Rm>

**is equivalent to**

MOV{<c>} {<q>} <Rd>, <Rm>, RRX

**and is always the preferred disassembly.**

#### Assembler symbols for all encodings

- <c> See [C1.2.5 Standard assembler syntax fields](#) on page 309.
- <q> See [C1.2.5 Standard assembler syntax fields](#) on page 309.
- <Rd> Is the general-purpose destination register, encoded in the "Rd" field.
- <Rm> Is the general-purpose source register, encoded in the "Rm" field.

#### Operation for all encodings

The description of [MOV \(register\)](#) gives the operational pseudocode for this instruction.

### C2.4.136 RRXS

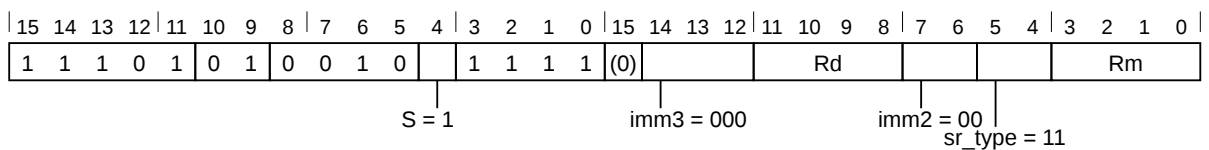
Rotate Right with Extend, Setting flags. Rotate Right with Extend, Setting flags shifts a register value right by one bit, shifting the Carry flag into bit[31] and bit[0] into the Carry flag, writes the result to the destination register and updates the condition flags (other than Carry) based on the result.

This instruction is an alias of the [MOV \(register\)](#) instruction. This means that:

- The encodings in this description are named to match the encodings of [MOV \(register\)](#).
- The description of [MOV \(register\)](#) gives the operational pseudocode for this instruction.

#### T3

*Armv8-M Main Extension only*



#### MOVS, rotate right with extend variant

RRXS {<c>} {<q>} {<Rd>}, {<Rm>}

**is equivalent to**

MOVS {<c>} {<q>} <Rd>, <Rm>, RRX

**and is always the preferred disassembly.**

#### Assembler symbols for all encodings

- <c> See [C1.2.5 Standard assembler syntax fields](#) on page 309.
- <q> See [C1.2.5 Standard assembler syntax fields](#) on page 309.
- <Rd> Is the general-purpose destination register, encoded in the "Rd" field.
- <Rm> Is the general-purpose source register, encoded in the "Rm" field.

#### Operation for all encodings

The description of [MOV \(register\)](#) gives the operational pseudocode for this instruction.

### C2.4.137 RSB (immediate)

Reverse Subtract (immediate). Reverse Subtract (immediate) subtracts a register value from an immediate value, and writes the result to the destination register. It can optionally update the condition flags based on the result.

#### T1

Armv8-M

|    |    |    |    |    |    |   |   |   |   |    |    |   |   |   |   |
|----|----|----|----|----|----|---|---|---|---|----|----|---|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5  | 4  | 3 | 2 | 1 | 0 |
| 0  | 1  | 0  | 0  | 0  | 0  | 1 | 0 | 0 | 1 | Rn | Rd |   |   |   |   |

#### T1 variant

```
RSB<c>{<q>} {<Rd>, }<Rn>, #0
// Inside IT block
RSBS{<q>} {<Rd>, }<Rn>, #0
// Outside IT block
```

#### Decode for this encoding

```
1 d = UInt(Rd); n = UInt(Rn); setflags = !InITBlock(); imm32 = Zeros(32); // immediate = #0
```

#### T2

Armv8-M Main Extension only

|    |    |    |    |    |    |   |   |   |   |   |   |    |   |      |    |      |    |    |    |    |    |   |   |   |   |   |   |   |   |   |   |
|----|----|----|----|----|----|---|---|---|---|---|---|----|---|------|----|------|----|----|----|----|----|---|---|---|---|---|---|---|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3  | 2 | 1    | 0  | 15   | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 1  | 1  | 1  | 1  | 0  | i  | 0 | 1 | 1 | 1 | 0 | S | Rn | 0 | imm3 | Rd | imm8 |    |    |    |    |    |   |   |   |   |   |   |   |   |   |   |

#### RSB variant

Applies when **S** == 0.

```
RSB<c>.W {<Rd>, } <Rn>, #0
// Inside IT block
RSB{<c>}{<q>} {<Rd>, } <Rn>, #<const>
```

#### RSBS variant

Applies when **S** == 1.

```
RSBS.W {<Rd>, } <Rn>, #0
// Outside IT block
RSBS{<c>}{<q>} {<Rd>, } <Rn>, #<const>
```

#### Decode for this encoding

```
1 if !HaveMainExt() then UNDEFINED;
2 d = UInt(Rd); n = UInt(Rn); setflags = (S == '1'); imm32 = T32ExpandImm(i:imm3:imm8);
3 if d IN {13,15} || n IN {13,15} then UNPREDICTABLE;
```

#### Assembler symbols for all encodings

<c> See [C1.2.5 Standard assembler syntax fields](#) on page 309.  
 <q> See [C1.2.5 Standard assembler syntax fields](#) on page 309.  
 <Rd> Is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as <Rn>.

<Rn> Is the general-purpose source register, encoded in the "Rn" field.  
<const> Is an immediate value derived from the 12-bit immediate that is encoded in the 'i:imm3:imm8' field. See [C1.5 Modified immediate constants](#) on page 324 for the range of values.

### Operation for all encodings

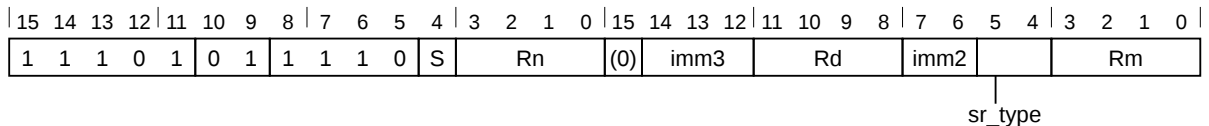
```
1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 (result, carry, overflow) = AddWithCarry(NOT(R[n]), imm32, '1');
4 R[d] = result;
5 if setflags then
6 APSR.N = result[31];
7 APSR.Z = IsZeroBit(result);
8 APSR.C = carry;
9 APSR.V = overflow;
```

### C2.4.138 RSB (register)

Reverse Subtract (register). Reverse Subtract (register) subtracts a register value from an optionally-shifted register value, and writes the result to the destination register. It can optionally update the condition flags based on the result.

#### T1

Armv8-M Main Extension only



#### RSB, rotate right with extend variant

Applies when **S == 0** && **imm3 == 000** && **imm2 == 00** && **sr\_type == 11**.

RSB{<c>}{<q>} {<Rd>}, <Rn>, <Rm>, RRX

#### RSB, shift or rotate by value variant

Applies when **S == 0** && **!(imm3 == 000 && imm2 == 00 && sr\_type == 11)**.

RSB{<c>}{<q>} {<Rd>}, <Rn>, <Rm> {, <shift> #<amount>}

#### RSBS, rotate right with extend variant

Applies when **S == 1** && **imm3 == 000** && **imm2 == 00** && **sr\_type == 11**.

RSBS{<c>}{<q>} {<Rd>}, <Rn>, <Rm>, RRX

#### RSBS, shift or rotate by value variant

Applies when **S == 1** && **!(imm3 == 000 && imm2 == 00 && sr\_type == 11)**.

RSBS{<c>}{<q>} {<Rd>}, <Rn>, <Rm> {, <shift> #<amount>}

#### Decode for this encoding

```

1 if !HaveMainExt() then UNDEFINED;
2 d = UInt(Rd); n = UInt(Rn); m = UInt(Rm); setflags = (S == '1');
3 (shift_t, shift_n) = DecodeImmShift(sr_type, imm3:imm2);
4 if d IN {13,15} || n IN {13,15} || m IN {13,15} then UNPREDICTABLE;
```

#### Assembler symbols for all encodings

- <c> See [C1.2.5 Standard assembler syntax fields](#) on page 309.
- <q> See [C1.2.5 Standard assembler syntax fields](#) on page 309.
- <Rd> Is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as <Rn>.
- <Rn> Is the first general-purpose source register, encoded in the "Rn" field.
- <Rm> Is the second general-purpose source register, encoded in the "Rm" field.
- <shift> Is the type of shift to be applied to the second source register, encoded in the "sr\_type" field. It can have the following values:
  - LSL when sr\_type = 00
  - LSR when sr\_type = 01
  - ASR when sr\_type = 10
  - ROR when sr\_type = 11

<amount> Is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR) encoded in the "imm3:imm2" field as <amount> modulo 32.

### Operation for all encodings

```
1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 shifted = Shift(R[m], shift_t, shift_n, APSR.C);
4 (result, carry, overflow) = AddWithCarry(NOT(R[n]), shifted, '1');
5 R[d] = result;
6 if setflags then
7 APSR.Z = IsZeroBit(result);
8 APSR.N = result[31];
9 APSR.C = carry;
10 APSR.V = overflow;
```



### C2.4.139 SADD16

Signed Add 16. Signed Add 16 performs two 16-bit signed integer additions, and writes the results to the destination register. It sets the APSR.GE bits according to the results of the additions.

#### T1

*Armv8-M DSP Extension only*

|    |    |    |    |    |    |   |   |   |   |   |   |    |   |   |   |    |    |    |    |    |    |   |   |   |   |   |   |    |   |   |   |
|----|----|----|----|----|----|---|---|---|---|---|---|----|---|---|---|----|----|----|----|----|----|---|---|---|---|---|---|----|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3  | 2 | 1 | 0 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3  | 2 | 1 | 0 |
| 1  | 1  | 1  | 1  | 1  | 0  | 1 | 0 | 1 | 0 | 0 | 1 | Rn |   |   |   | 1  | 1  | 1  | 1  | Rd |    |   |   | 0 | 0 | 0 | 0 | Rm |   |   |   |

#### T1 variant

SADD16{<c>}{<q>} {<Rd>}, {<Rn>}, <Rm>

#### Decode for this encoding

```

1 if !HaveDSPExt() then UNDEFINED;
2 d = UInt(Rd); n = UInt(Rn); m = UInt(Rm);
3 if d IN {13,15} || n IN {13,15} || m IN {13,15} then UNPREDICTABLE;

```

#### Assembler symbols for all encodings

<c> See [C1.2.5 Standard assembler syntax fields](#) on page 309.  
 <q> See [C1.2.5 Standard assembler syntax fields](#) on page 309.  
 <Rd> Is the general-purpose destination register, encoded in the "Rd" field.  
 <Rn> Is the first general-purpose source register, encoded in the "Rn" field.  
 <Rm> Is the second general-purpose source register, encoded in the "Rm" field.

#### Operation for all encodings

```

1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 sum1 = SInt(R[n][15:0]) + SInt(R[m][15:0]);
4 sum2 = SInt(R[n][31:16]) + SInt(R[m][31:16]);
5 R[d] = sum2[15:0] : sum1[15:0];
6 APSR.GE[1:0] = if sum1 >= 0 then '11' else '00';
7 APSR.GE[3:2] = if sum2 >= 0 then '11' else '00';

```

## C2.4.140 SADD8

Signed Add 8. Signed Add 8 performs four 8-bit signed integer additions, and writes the results to the destination register. It sets the APSR.GE bits according to the results of the additions.

### T1

Armv8-M DSP Extension only

|    |    |    |    |    |    |   |   |   |   |   |   |    |   |   |   |    |    |    |    |    |    |   |   |   |   |   |   |    |   |   |   |
|----|----|----|----|----|----|---|---|---|---|---|---|----|---|---|---|----|----|----|----|----|----|---|---|---|---|---|---|----|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3  | 2 | 1 | 0 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3  | 2 | 1 | 0 |
| 1  | 1  | 1  | 1  | 1  | 0  | 1 | 0 | 1 | 0 | 0 | 0 | Rn |   |   |   | 1  | 1  | 1  | 1  | Rd |    |   |   | 0 | 0 | 0 | 0 | Rm |   |   |   |

### T1 variant

SADD8 {<c>} {<q>} {<Rd>}, {<Rn>}, {<Rm>}

### Decode for this encoding

```

1 if !HaveDSPExt() then UNDEFINED;
2 d = UInt(Rd); n = UInt(Rn); m = UInt(Rm);
3 if d IN {13,15} || n IN {13,15} || m IN {13,15} then UNPREDICTABLE;

```

### Assembler symbols for all encodings

<c> See [C1.2.5 Standard assembler syntax fields](#) on page 309.  
 <q> See [C1.2.5 Standard assembler syntax fields](#) on page 309.  
 <Rd> Is the general-purpose destination register, encoded in the "Rd" field.  
 <Rn> Is the first general-purpose source register, encoded in the "Rn" field.  
 <Rm> Is the second general-purpose source register, encoded in the "Rm" field.

### Operation for all encodings

```

1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 sum1 = SInt(R[n][7:0]) + SInt(R[m][7:0]);
4 sum2 = SInt(R[n][15:8]) + SInt(R[m][15:8]);
5 sum3 = SInt(R[n][23:16]) + SInt(R[m][23:16]);
6 sum4 = SInt(R[n][31:24]) + SInt(R[m][31:24]);
7 R[d] = sum4[7:0] : sum3[7:0] : sum2[7:0] : sum1[7:0];
8 APSR.GE[0] = if sum1 >= 0 then '1' else '0';
9 APSR.GE[1] = if sum2 >= 0 then '1' else '0';
10 APSR.GE[2] = if sum3 >= 0 then '1' else '0';
11 APSR.GE[3] = if sum4 >= 0 then '1' else '0';

```

### C2.4.141 SASX

Signed Add and Subtract with Exchange. Signed Add and Subtract with Exchange exchanges the two halfwords of the second operand, performs one 16-bit integer addition and one 16-bit subtraction, and writes the results to the destination register. It sets the APSR.GE bits according to the results.

#### T1

Armv8-M DSP Extension only

|    |    |    |    |    |    |   |   |   |   |   |   |    |   |   |   |    |    |    |    |    |    |   |   |   |   |   |   |    |   |   |   |
|----|----|----|----|----|----|---|---|---|---|---|---|----|---|---|---|----|----|----|----|----|----|---|---|---|---|---|---|----|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3  | 2 | 1 | 0 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3  | 2 | 1 | 0 |
| 1  | 1  | 1  | 1  | 1  | 0  | 1 | 0 | 1 | 0 | 1 | 0 | Rn |   |   |   | 1  | 1  | 1  | 1  | Rd |    |   |   | 0 | 0 | 0 | 0 | Rm |   |   |   |

#### T1 variant

SASX{<c>}{<q>} {<Rd>}, {<Rn>}, <Rm>

#### Decode for this encoding

```

1 if !HaveDSPExt() then UNDEFINED;
2 d = UInt(Rd); n = UInt(Rn); m = UInt(Rm);
3 if d IN {13,15} || n IN {13,15} || m IN {13,15} then UNPREDICTABLE;

```

#### Assembler symbols for all encodings

<c> See [C1.2.5 Standard assembler syntax fields](#) on page 309.  
 <q> See [C1.2.5 Standard assembler syntax fields](#) on page 309.  
 <Rd> Is the general-purpose destination register, encoded in the "Rd" field.  
 <Rn> Is the first general-purpose source register, encoded in the "Rn" field.  
 <Rm> Is the second general-purpose source register, encoded in the "Rm" field.

#### Operation for all encodings

```

1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 diff = SInt(R[n][15:0]) - SInt(R[m][31:16]);
4 sum = SInt(R[n][31:16]) + SInt(R[m][15:0]);
5 R[d] = sum[15:0] : diff[15:0];
6 APSR.GE[1:0] = if diff >= 0 then '11' else '00';
7 APSR.GE[3:2] = if sum >= 0 then '11' else '00';

```

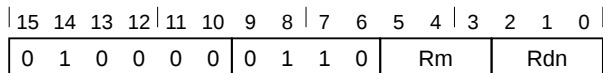


### C2.4.143 SBC (register)

Subtract with Carry (register). Subtract with Carry (register) subtracts an optionally-shifted register value and the value of NOT(Carry flag) from a register value, and writes the result to the destination register. It can optionally update the condition flags based on the result.

#### T1

Armv8-M



#### T1 variant

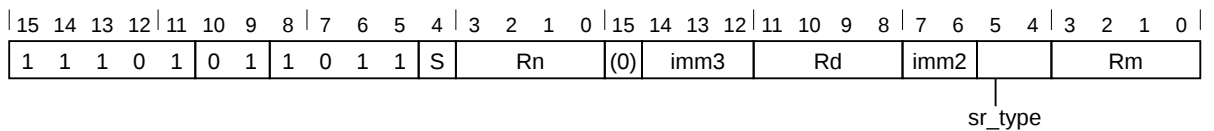
```
SBC<c>{<q>} {<Rdn>, } <Rdn>, <Rm>
 // Inside IT block
SBCS{<q>} {<Rdn>, } <Rdn>, <Rm>
 // Outside IT block
```

#### Decode for this encoding

```
1 d = UInt(Rdn); n = UInt(Rdn); m = UInt(Rm); setflags = !InITBlock();
2 (shift_t, shift_n) = (SRTYPE_LSL, 0);
```

#### T2

Armv8-M Main Extension only



#### SBC, rotate right with extend variant

Applies when **S == 0** && **imm3 == 000** && **imm2 == 00** && **sr\_type == 11**.

```
SBC{<c>}{<q>} {<Rd>, } <Rn>, <Rm>, RRX
```

#### SBC, shift or rotate by value variant

Applies when **S == 0** && **!(imm3 == 000 && imm2 == 00 && sr\_type == 11)**.

```
SBC<c>.W {<Rd>, } <Rn>, <Rm>
 // Inside IT block, and <Rd>, <Rn>, <Rm> can be represented in T1
SBC{<c>}{<q>} {<Rd>, } <Rn>, <Rm> {, <shift> #<amount>}
```

#### SBCS, rotate right with extend variant

Applies when **S == 1** && **imm3 == 000** && **imm2 == 00** && **sr\_type == 11**.

```
SBCS{<c>}{<q>} {<Rd>, } <Rn>, <Rm>, RRX
```

#### SBCS, shift or rotate by value variant

Applies when **S == 1** && **!(imm3 == 000 && imm2 == 00 && sr\_type == 11)**.

```
SBCS.W {<Rd>, } <Rn>, <Rm>
 // Outside IT block, and <Rd>, <Rn>, <Rm> can be represented in T1
SBCS{<c>}{<q>} {<Rd>, } <Rn>, <Rm> {, <shift> #<amount>}
```

### Decode for this encoding

```
1 if !HaveMainExt() then UNDEFINED;
2 d = UInt(Rd); n = UInt(Rn); m = UInt(Rm); setflags = (S == '1');
3 (shift_t, shift_n) = DecodeImmShift(sr_type, imm3:imm2);
4 if d IN {13,15} || n IN {13,15} || m IN {13,15} then UNPREDICTABLE;
```

### Assembler symbols for all encodings

<c> See [C1.2.5 Standard assembler syntax fields](#) on page 309.

<q> See [C1.2.5 Standard assembler syntax fields](#) on page 309.

<Rdn> Is the first general-purpose source register and the destination register, encoded in the "Rdn" field.

<Rd> Is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as <Rn>.

<Rn> Is the first general-purpose source register, encoded in the "Rn" field.

<Rm> Is the second general-purpose source register, encoded in the "Rm" field.

<shift> Is the type of shift to be applied to the second source register, encoded in the "sr\_type" field. It can have the following values:

- LSL when sr\_type = 00
- LSR when sr\_type = 01
- ASR when sr\_type = 10
- ROR when sr\_type = 11

<amount> Is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR) encoded in the "imm3:imm2" field as <amount> modulo 32.

### Operation for all encodings

```
1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 shifted = Shift(R[m], shift_t, shift_n, APSR.C);
4 (result, carry, overflow) = AddWithCarry(R[n], NOT(shifted), APSR.C);
5 R[d] = result;
6 if setflags then
7 APSR.N = result[31];
8 APSR.Z = IsZeroBit(result);
9 APSR.C = carry;
10 APSR.V = overflow;
```

## C2.4.144 SBFX

Signed Bit Field Extract. Signed Bit Field Extract extracts any number of adjacent bits at any position from one register, sign extends them to 32 bits, and writes the result to the destination register.

### T1

Armv8-M Main Extension only

|    |    |    |    |    |     |   |   |   |   |   |   |    |   |   |   |    |      |    |    |    |    |   |   |   |      |     |         |   |   |   |   |
|----|----|----|----|----|-----|---|---|---|---|---|---|----|---|---|---|----|------|----|----|----|----|---|---|---|------|-----|---------|---|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10  | 9 | 8 | 7 | 6 | 5 | 4 | 3  | 2 | 1 | 0 | 15 | 14   | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6    | 5   | 4       | 3 | 2 | 1 | 0 |
| 1  | 1  | 1  | 1  | 0  | (0) | 1 | 1 | 0 | 1 | 0 | 0 | Rn |   |   |   | 0  | imm3 |    |    |    | Rd |   |   |   | imm2 | (0) | widthm1 |   |   |   |   |

### T1 variant

SBFX{<c>}{<q>} <Rd>, <Rn>, #<lsb>, #<width>

### Decode for this encoding

```

1 if !HaveMainExt() then UNDEFINED;
2 d = UInt(Rd); n = UInt(Rn);
3 lsbit = UInt(imm3:imm2); widthminus1 = UInt(widthm1);
4 msbit = lsbit + widthminus1;
5 if msbit > 31 then UNPREDICTABLE;
6 if d IN {13,15} || n IN {13,15} then UNPREDICTABLE;

```

### CONSTRAINED UNPREDICTABLE behavior

If `msbit > 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The value in the destination register is UNKNOWN.

### Assembler symbols for all encodings

|         |                                                                                                                      |
|---------|----------------------------------------------------------------------------------------------------------------------|
| <c>     | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                                             |
| <q>     | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                                             |
| <Rd>    | Is the general-purpose destination register, encoded in the "Rd" field.                                              |
| <Rn>    | Is the general-purpose source register, encoded in the "Rn" field.                                                   |
| <lsb>   | Is the bit number of the least significant bit in the field, in the range 0 to 31, encoded in the "imm3:imm2" field. |
| <width> | Is the width of the field, in the range 1 to 32-<lsb>, encoded in the "widthm1" field as <width>-1.                  |

### Operation for all encodings

```

1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 if msbit <= 31 then
4 R[d] = SignExtend(R[n][msbit:lsbit], 32);
5 else
6 R[d] = bits(32) UNKNOWN;

```

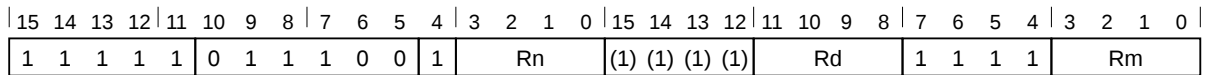
### C2.4.145 SDIV

Signed Divide. Signed Divide divides a 32-bit signed integer register value by a 32-bit signed integer register value and writes the result to the destination register. The condition code flags are not affected.

If  $R[n] == 0 \times 80000000$  ( $-2^{31}$ ) and  $R[m] == 0 \times \text{FFFFFFFF}$  ( $-1$ ), the result of the division is  $0 \times 80000000$ .

#### T1

Armv8-M



#### T1 variant

SDIV{<c>}{<q>} {<Rd>, } <Rn>, <Rm>

#### Decode for this encoding

```
1 d = UInt(Rd); n = UInt(Rn); m = UInt(Rm);
2 if d IN {13,15} || n IN {13,15} || m IN {13,15} then UNPREDICTABLE;
```

#### Assembler symbols for all encodings

- <c> See [C1.2.5 Standard assembler syntax fields](#) on page 309.
- <q> See [C1.2.5 Standard assembler syntax fields](#) on page 309.
- <Rd> Is the general-purpose destination register, encoded in the "Rd" field.
- <Rn> Is the first general-purpose source register holding the dividend, encoded in the "Rn" field.
- <Rm> Is the second general-purpose source register holding the divisor, encoded in the "Rm" field.

#### Operation for all encodings

```
1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 if SInt(R[m]) == 0 then
4 if IntegerZeroDivideTrappingEnabled() then
5 GenerateIntegerZeroDivide();
6 else
7 result = 0;
8 else
9 result = RoundTowardsZero(Real(SInt(R[n])) / Real(SInt(R[m])));
10 R[d] = result[31:0];
```



## C2.4.146 SEL

Select Bytes. Select Bytes selects each byte of its result from either its first operand or its second operand, according to the values of the GE flags.

### T1

Armv8-M DSP Extension only

|    |    |    |    |    |    |   |   |   |   |   |   |    |   |   |   |    |    |    |    |    |    |   |   |   |   |   |   |    |   |   |   |
|----|----|----|----|----|----|---|---|---|---|---|---|----|---|---|---|----|----|----|----|----|----|---|---|---|---|---|---|----|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3  | 2 | 1 | 0 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3  | 2 | 1 | 0 |
| 1  | 1  | 1  | 1  | 1  | 0  | 1 | 0 | 1 | 0 | 1 | 0 | Rn |   |   |   | 1  | 1  | 1  | 1  | Rd |    |   |   | 1 | 0 | 0 | 0 | Rm |   |   |   |

### T1 variant

SEL{<c>}{<q>} {<Rd>}, {<Rn>}, {<Rm>}

### Decode for this encoding

```

1 if !HaveDSPExt() then UNDEFINED;
2 d = UInt(Rd); n = UInt(Rn); m = UInt(Rm);
3 if d IN {13,15} || n IN {13,15} || m IN {13,15} then UNPREDICTABLE;

```

### Assembler symbols for all encodings

<c> See [C1.2.5 Standard assembler syntax fields](#) on page 309.  
 <q> See [C1.2.5 Standard assembler syntax fields](#) on page 309.  
 <Rd> Is the general-purpose destination register, encoded in the "Rd" field.  
 <Rn> Is the first general-purpose source register, encoded in the "Rn" field.  
 <Rm> Is the second general-purpose source register, encoded in the "Rm" field.

### Operation for all encodings

```

1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 bits(32) result;
4 result[7:0] = if APSR.GE[0] == '1' then R[n][7:0] else R[m][7:0];
5 result[15:8] = if APSR.GE[1] == '1' then R[n][15:8] else R[m][15:8];
6 result[23:16] = if APSR.GE[2] == '1' then R[n][23:16] else R[m][23:16];
7 result[31:24] = if APSR.GE[3] == '1' then R[n][31:24] else R[m][31:24];
8 R[d] = result;

```

### C2.4.147 SEV

Send Event. Send Event is a hint instruction. It causes an event to be signaled to all PEs within the multiprocessor system.

This is a NOP-compatible hint.

#### T1

*Armv8-M*

|    |    |    |    |    |    |   |   |   |   |   |   |   |   |   |   |
|----|----|----|----|----|----|---|---|---|---|---|---|---|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 1  | 0  | 1  | 1  | 1  | 1  | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |

#### T1 variant

SEV{<c>}{<q>}

#### Decode for this encoding

```
1 // No additional decoding required
```

#### T2

*Armv8-M Main Extension only*

|    |    |    |    |    |    |   |   |   |   |   |   |     |     |     |     |    |    |     |    |     |    |   |   |   |   |   |   |   |   |   |   |
|----|----|----|----|----|----|---|---|---|---|---|---|-----|-----|-----|-----|----|----|-----|----|-----|----|---|---|---|---|---|---|---|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3   | 2   | 1   | 0   | 15 | 14 | 13  | 12 | 11  | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 1  | 1  | 1  | 1  | 0  | 0  | 1 | 1 | 1 | 0 | 1 | 0 | (1) | (1) | (1) | (1) | 1  | 0  | (0) | 0  | (0) | 0  | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |   |

#### T2 variant

SEV{<c>}.W

#### Decode for this encoding

```
1 if !HaveMainExt() then UNDEFINED;
2 // No additional decoding required
```

#### Assembler symbols for all encodings

<c> See [C1.2.5 Standard assembler syntax fields](#) on page 309.  
 <q> See [C1.2.5 Standard assembler syntax fields](#) on page 309.

#### Operation for all encodings

```
1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 SendEvent();
```

## C2.4.148 SG

Secure Gateway. Secure Gateway marks a valid branch target for branches from Non-secure code that call Secure code.

This instruction sets the Security state to Secure if its address is in Secure memory. If the address of this instruction is in Non-secure memory, the instruction behaves as a NOP.

If the PE was previously in Non-secure state:

- This instruction sets bit[0] of LR to 0, to indicate that the return address will cause a transition from Secure to Non-secure state.
- If the Floating-point Extension is implemented, this instruction marks Secure floating-point state as inactive, by setting CONTROL\_S.SFPA to 0. This indicates that the floating-point registers do not contain active state that belongs to the Secure state.

If the Security Extension is not implemented, this instruction behaves as a NOP.

**SG** is an unconditional instruction and executes as such both inside and outside an **IT** instruction block. Arm recommends that software does not place **SG** inside an **IT** instruction block.

### T1

Armv8-M

|    |    |    |    |    |    |   |   |   |   |   |   |   |   |   |   |    |    |    |    |    |    |   |   |   |   |   |   |   |   |   |   |
|----|----|----|----|----|----|---|---|---|---|---|---|---|---|---|---|----|----|----|----|----|----|---|---|---|---|---|---|---|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 1  | 1  | 1  | 0  | 1  | 0  | 0 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1  | 1  | 1  | 0  | 1  | 0  | 0 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 1 |   |

### T1 variant

SG{<q>}

### Decode for this encoding

```
1 // No encoding specific operations
```

### Assembler symbols for all encodings

<q> See [C1.2.5 Standard assembler syntax fields](#) on page 309.

### Operation for all encodings

```
1 EncodingSpecificOperations();
2
3 if HaveSecurityExt() then
4 sAttributes = SecurityCheck(ThisInstrAddr(), TRUE, IsSecure());
5 if !sAttributes.ns then
6 if !IsSecure() then
7 LR[0] = '0';
8 if HaveFPEExt() then
9 CONTROL_S.SFPA = '0';
10 CurrentState = SecurityState_Secure;
11 // IT/ICI bits cleared to prevent Non-secure code interfering with
12 // Secure execution
13 if HaveMainExt() then
14 ITSTATE = Zeros(8);
```

## C2.4.149 SHADD16

Signed Halving Add 16. Signed Halving Add 16 performs two signed 16-bit integer additions, halves the results, and writes the results to the destination register.

### T1

*Armv8-M DSP Extension only*

|    |    |    |    |    |    |   |   |   |   |   |   |    |   |   |   |    |    |    |    |    |    |   |   |   |   |   |   |    |   |   |   |
|----|----|----|----|----|----|---|---|---|---|---|---|----|---|---|---|----|----|----|----|----|----|---|---|---|---|---|---|----|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3  | 2 | 1 | 0 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3  | 2 | 1 | 0 |
| 1  | 1  | 1  | 1  | 1  | 0  | 1 | 0 | 1 | 0 | 0 | 1 | Rn |   |   |   | 1  | 1  | 1  | 1  | Rd |    |   |   | 0 | 0 | 1 | 0 | Rm |   |   |   |

### T1 variant

SHADD16{<c>}{<q>} {<Rd>}, {<Rn>}, {<Rm>}

### Decode for this encoding

```

1 if !HaveDSPExt() then UNDEFINED;
2 d = UInt(Rd); n = UInt(Rn); m = UInt(Rm);
3 if d IN {13,15} || n IN {13,15} || m IN {13,15} then UNPREDICTABLE;

```

### Assembler symbols for all encodings

<c> See [C1.2.5 Standard assembler syntax fields](#) on page 309.  
 <q> See [C1.2.5 Standard assembler syntax fields](#) on page 309.  
 <Rd> Is the general-purpose destination register, encoded in the "Rd" field.  
 <Rn> Is the first general-purpose source register, encoded in the "Rn" field.  
 <Rm> Is the second general-purpose source register, encoded in the "Rm" field.

### Operation for all encodings

```

1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 sum1 = SInt(R[n][15:0]) + SInt(R[m][15:0]);
4 sum2 = SInt(R[n][31:16]) + SInt(R[m][31:16]);
5 R[d] = sum2[16:1] : sum1[16:1];

```

## C2.4.150 SHADD8

Signed Halving Add 8. Signed Halving Add 8 performs four signed 8-bit integer additions, halves the results, and writes the results to the destination register.

### T1

*Armv8-M DSP Extension only*

|    |    |    |    |    |    |   |   |   |   |   |   |    |   |   |   |    |    |    |    |    |    |   |   |   |   |   |   |    |   |   |   |
|----|----|----|----|----|----|---|---|---|---|---|---|----|---|---|---|----|----|----|----|----|----|---|---|---|---|---|---|----|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3  | 2 | 1 | 0 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3  | 2 | 1 | 0 |
| 1  | 1  | 1  | 1  | 1  | 0  | 1 | 0 | 1 | 0 | 0 | 0 | Rn |   |   |   | 1  | 1  | 1  | 1  | Rd |    |   |   | 0 | 0 | 1 | 0 | Rm |   |   |   |

### T1 variant

SHADD8{<c>}{<q>} {<Rd>}, {<Rn>}, <Rm>

### Decode for this encoding

```

1 if !HaveDSPExt() then UNDEFINED;
2 d = UInt(Rd); n = UInt(Rn); m = UInt(Rm);
3 if d IN {13,15} || n IN {13,15} || m IN {13,15} then UNPREDICTABLE;
```

### Assembler symbols for all encodings

<c> See [C1.2.5 Standard assembler syntax fields](#) on page 309.  
 <q> See [C1.2.5 Standard assembler syntax fields](#) on page 309.  
 <Rd> Is the general-purpose destination register, encoded in the "Rd" field.  
 <Rn> Is the first general-purpose source register, encoded in the "Rn" field.  
 <Rm> Is the second general-purpose source register, encoded in the "Rm" field.

### Operation for all encodings

```

1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 sum1 = SInt(R[n][7:0]) + SInt(R[m][7:0]);
4 sum2 = SInt(R[n][15:8]) + SInt(R[m][15:8]);
5 sum3 = SInt(R[n][23:16]) + SInt(R[m][23:16]);
6 sum4 = SInt(R[n][31:24]) + SInt(R[m][31:24]);
7 R[d] = sum4[8:1] : sum3[8:1] : sum2[8:1] : sum1[8:1];
```

## C2.4.151 SHASX

Signed Halving Add and Subtract with Exchange. Signed Halving Add and Subtract with Exchange exchanges the two halfwords of the second operand, performs one signed 16-bit integer addition and one signed 16-bit subtraction, halves the results, and writes the results to the destination register.

### T1

Armv8-M DSP Extension only

|    |    |    |    |    |    |   |   |   |   |   |   |    |   |   |   |    |    |    |    |    |    |   |   |   |   |   |   |    |   |   |   |
|----|----|----|----|----|----|---|---|---|---|---|---|----|---|---|---|----|----|----|----|----|----|---|---|---|---|---|---|----|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3  | 2 | 1 | 0 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3  | 2 | 1 | 0 |
| 1  | 1  | 1  | 1  | 1  | 0  | 1 | 0 | 1 | 0 | 1 | 0 | Rn |   |   |   | 1  | 1  | 1  | 1  | Rd |    |   |   | 0 | 0 | 1 | 0 | Rm |   |   |   |

### T1 variant

SHASX{<c>}{<q>} {<Rd>}, {<Rn>}, {<Rm>}

### Decode for this encoding

```

1 if !HaveDSPExt() then UNDEFINED;
2 d = UInt(Rd); n = UInt(Rn); m = UInt(Rm);
3 if d IN {13,15} || n IN {13,15} || m IN {13,15} then UNPREDICTABLE;

```

### Assembler symbols for all encodings

<c> See [C1.2.5 Standard assembler syntax fields](#) on page 309.  
 <q> See [C1.2.5 Standard assembler syntax fields](#) on page 309.  
 <Rd> Is the general-purpose destination register, encoded in the "Rd" field.  
 <Rn> Is the first general-purpose source register, encoded in the "Rn" field.  
 <Rm> Is the second general-purpose source register, encoded in the "Rm" field.

### Operation for all encodings

```

1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 diff = SInt(R[n][15:0]) - SInt(R[m][31:16]);
4 sum = SInt(R[n][31:16]) + SInt(R[m][15:0]);
5 R[d] = sum[16:1] : diff[16:1];

```

## C2.4.152 SHSAX

Signed Halving Subtract and Add with Exchange. Signed Halving Subtract and Add with Exchange exchanges the two halfwords of the second operand, performs one signed 16-bit integer subtraction and one signed 16-bit addition, halves the results, and writes the results to the destination register.

### T1

Armv8-M DSP Extension only

|    |    |    |    |    |    |   |   |   |   |   |   |    |   |   |   |    |    |    |    |    |    |   |   |   |   |   |   |    |   |   |   |
|----|----|----|----|----|----|---|---|---|---|---|---|----|---|---|---|----|----|----|----|----|----|---|---|---|---|---|---|----|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3  | 2 | 1 | 0 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3  | 2 | 1 | 0 |
| 1  | 1  | 1  | 1  | 1  | 0  | 1 | 0 | 1 | 1 | 1 | 0 | Rn |   |   |   | 1  | 1  | 1  | 1  | Rd |    |   |   | 0 | 0 | 1 | 0 | Rm |   |   |   |

### T1 variant

SHSAX{<c>}{<q>} {<Rd>}, {<Rn>}, {<Rm>}

### Decode for this encoding

```

1 if !HaveDSPExt() then UNDEFINED;
2 d = UInt(Rd); n = UInt(Rn); m = UInt(Rm);
3 if d IN {13,15} || n IN {13,15} || m IN {13,15} then UNPREDICTABLE;

```

### Assembler symbols for all encodings

<c> See [C1.2.5 Standard assembler syntax fields](#) on page 309.  
 <q> See [C1.2.5 Standard assembler syntax fields](#) on page 309.  
 <Rd> Is the general-purpose destination register, encoded in the "Rd" field.  
 <Rn> Is the first general-purpose source register, encoded in the "Rn" field.  
 <Rm> Is the second general-purpose source register, encoded in the "Rm" field.

### Operation for all encodings

```

1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 sum = SInt(R[n][15:0]) + SInt(R[m][31:16]);
4 diff = SInt(R[n][31:16]) - SInt(R[m][15:0]);
5 R[d] = diff[16:1] : sum[16:1];

```

### C2.4.153 SHSUB16

Signed Halving Subtract 16. Signed Halving Subtract 16 performs two signed 16-bit integer subtractions, halves the results, and writes the results to the destination register.

#### T1

Armv8-M DSP Extension only

|    |    |    |    |    |    |   |   |   |   |   |   |    |   |   |   |    |    |    |    |    |    |   |   |   |   |   |   |    |   |   |   |
|----|----|----|----|----|----|---|---|---|---|---|---|----|---|---|---|----|----|----|----|----|----|---|---|---|---|---|---|----|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3  | 2 | 1 | 0 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3  | 2 | 1 | 0 |
| 1  | 1  | 1  | 1  | 1  | 0  | 1 | 0 | 1 | 1 | 0 | 1 | Rn |   |   |   | 1  | 1  | 1  | 1  | Rd |    |   |   | 0 | 0 | 1 | 0 | Rm |   |   |   |

#### T1 variant

SHSUB16{<c>}{<q>} {<Rd>}, {<Rn>}, {<Rm>}

#### Decode for this encoding

```

1 if !HaveDSPExt() then UNDEFINED;
2 d = UInt(Rd); n = UInt(Rn); m = UInt(Rm);
3 if d IN {13,15} || n IN {13,15} || m IN {13,15} then UNPREDICTABLE;

```

#### Assembler symbols for all encodings

<c> See [C1.2.5 Standard assembler syntax fields](#) on page 309.  
 <q> See [C1.2.5 Standard assembler syntax fields](#) on page 309.  
 <Rd> Is the general-purpose destination register, encoded in the "Rd" field.  
 <Rn> Is the first general-purpose source register, encoded in the "Rn" field.  
 <Rm> Is the second general-purpose source register, encoded in the "Rm" field.

#### Operation for all encodings

```

1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 diff1 = SInt(R[n][15:0]) - SInt(R[m][15:0]);
4 diff2 = SInt(R[n][31:16]) - SInt(R[m][31:16]);
5 R[d] = diff2[16:1] : diff1[16:1];

```



### C2.4.154 SHSUB8

Signed Halving Subtract 8. Signed Halving Subtract 8 performs four signed 8-bit integer subtractions, halves the results, and writes the results to the destination register.

#### T1

Armv8-M DSP Extension only

|    |    |    |    |    |    |   |   |   |   |   |   |    |   |   |   |    |    |    |    |    |    |   |   |   |   |   |   |    |   |   |   |
|----|----|----|----|----|----|---|---|---|---|---|---|----|---|---|---|----|----|----|----|----|----|---|---|---|---|---|---|----|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3  | 2 | 1 | 0 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3  | 2 | 1 | 0 |
| 1  | 1  | 1  | 1  | 1  | 0  | 1 | 0 | 1 | 1 | 0 | 0 | Rn |   |   |   | 1  | 1  | 1  | 1  | Rd |    |   |   | 0 | 0 | 1 | 0 | Rm |   |   |   |

#### T1 variant

SHSUB8{<c>}{<q>} {<Rd>}, {<Rn>}, <Rm>

#### Decode for this encoding

```

1 if !HaveDSPExt() then UNDEFINED;
2 d = UInt(Rd); n = UInt(Rn); m = UInt(Rm);
3 if d IN {13,15} || n IN {13,15} || m IN {13,15} then UNPREDICTABLE;

```

#### Assembler symbols for all encodings

<c> See [C1.2.5 Standard assembler syntax fields](#) on page 309.  
 <q> See [C1.2.5 Standard assembler syntax fields](#) on page 309.  
 <Rd> Is the general-purpose destination register, encoded in the "Rd" field.  
 <Rn> Is the first general-purpose source register, encoded in the "Rn" field.  
 <Rm> Is the second general-purpose source register, encoded in the "Rm" field.

#### Operation for all encodings

```

1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 diff1 = SInt(R[n][7:0]) - SInt(R[m][7:0]);
4 diff2 = SInt(R[n][15:8]) - SInt(R[m][15:8]);
5 diff3 = SInt(R[n][23:16]) - SInt(R[m][23:16]);
6 diff4 = SInt(R[n][31:24]) - SInt(R[m][31:24]);
7 R[d] = diff4[8:1] : diff3[8:1] : diff2[8:1] : diff1[8:1];

```

### C2.4.155 SMLABB, SMLABT, SMLATB, SMLATT

Signed Multiply Accumulate (halfwords). Signed Multiply Accumulate (halfwords) performs a signed multiply accumulate operation. The multiply acts on two signed 16-bit quantities, taken from either the bottom or the top half of their respective source registers. The other halves of these source registers are ignored. The 32-bit product is added to a 32-bit accumulate value and the result is written to the destination register.

If overflow occurs during the addition of the accumulate value, the instruction sets the Q flag in the APSR. It is not possible for overflow to occur during the multiplication.

#### T1

*Armv8-M DSP Extension only*

|    |    |    |    |    |    |   |   |   |   |   |   |    |   |   |   |            |    |    |    |    |    |   |   |   |   |   |   |    |   |   |   |
|----|----|----|----|----|----|---|---|---|---|---|---|----|---|---|---|------------|----|----|----|----|----|---|---|---|---|---|---|----|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3  | 2 | 1 | 0 | 15         | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3  | 2 | 1 | 0 |
| 1  | 1  | 1  | 1  | 1  | 0  | 1 | 1 | 0 | 0 | 0 | 1 | Rn |   |   |   | Ra != 1111 |    |    |    | Rd |    |   |   | 0 | 0 | N | M | Rm |   |   |   |

#### SMLABB variant

Applies when **N == 0** && **M == 0**.

SMLABB{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>

#### SMLABT variant

Applies when **N == 0** && **M == 1**.

SMLABT{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>

#### SMLATB variant

Applies when **N == 1** && **M == 0**.

SMLATB{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>

#### SMLATT variant

Applies when **N == 1** && **M == 1**.

SMLATT{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>

#### Decode for this encoding

```

1 if Ra == '1111' then SEE "SMULBB, SMULBT, SMULTB, SMULTT";
2 if !HaveDSPExt() then UNDEFINED;
3 d = UInt(Rd); n = UInt(Rn); m = UInt(Rm); a = UInt(Ra);
4 n_high = (N == '1'); m_high = (M == '1');
5 if d IN {13,15} || n IN {13,15} || m IN {13,15} || a == 13 then UNPREDICTABLE;
```

#### Assembler symbols for all encodings

|      |                                                                                                                                               |
|------|-----------------------------------------------------------------------------------------------------------------------------------------------|
| <c>  | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                                                                      |
| <q>  | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                                                                      |
| <Rd> | Is the general-purpose destination register, encoded in the "Rd" field.                                                                       |
| <Rn> | Is the first general-purpose source register holding the multiplicand in the bottom or top half (selected by <x>), encoded in the "Rn" field. |
| <Rm> | Is the second general-purpose source register holding the multiplier in the bottom or top half (selected by <y>), encoded in the "Rm" field.  |
| <Ra> | Is the third general-purpose source register holding the addend, encoded in the "Ra" field.                                                   |

**Operation for all encodings**

```
1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 operand1 = if n_high then R[n][31:16] else R[n][15:0];
4 operand2 = if m_high then R[m][31:16] else R[m][15:0];
5 result = SInt(operand1) * SInt(operand2) + SInt(R[a]);
6 R[d] = result[31:0];
7 if result != SInt(result[31:0]) then // Signed overflow
8 APSR.Q = '1';
```

### C2.4.156 SMLAD, SMLADX

Signed Multiply Accumulate Dual. Signed Multiply Accumulate Dual performs two signed 16-bit by 16-bit multiplications. It adds the products to a 32-bit accumulate operand.

Optionally, the instruction can exchange the halfwords of the second operand before performing the arithmetic.

This instruction sets the Q flag if the accumulate operation overflows. Overflow cannot occur during the multiplications.

#### T1

Armv8-M DSP Extension only

|    |    |    |    |    |    |   |   |   |   |   |   |    |   |   |   |            |    |    |    |    |    |   |   |   |   |   |   |    |   |   |   |
|----|----|----|----|----|----|---|---|---|---|---|---|----|---|---|---|------------|----|----|----|----|----|---|---|---|---|---|---|----|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3  | 2 | 1 | 0 | 15         | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3  | 2 | 1 | 0 |
| 1  | 1  | 1  | 1  | 1  | 0  | 1 | 1 | 0 | 0 | 1 | 0 | Rn |   |   |   | Ra != 1111 |    |    |    | Rd |    |   |   | 0 | 0 | 0 | M | Rm |   |   |   |

#### SMLAD variant

Applies when **M == 0**.

SMLAD{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>

#### SMLADX variant

Applies when **M == 1**.

SMLADX{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>

#### Decode for this encoding

```

1 if Ra == '1111' then SEE SMUAD;
2 if !HaveDSPEExt() then UNDEFINED;
3 d = UInt(Rd); n = UInt(Rn); m = UInt(Rm); a = UInt(Ra);
4 m_swap = (M == '1');
5 if d IN {13,15} || n IN {13,15} || m IN {13,15} || a == 13 then UNPREDICTABLE;

```

#### Assembler symbols for all encodings

<c> See [C1.2.5 Standard assembler syntax fields](#) on page 309.  
 <q> See [C1.2.5 Standard assembler syntax fields](#) on page 309.  
 <Rd> Is the general-purpose destination register, encoded in the "Rd" field.  
 <Rn> Is the first general-purpose source register, encoded in the "Rn" field.  
 <Rm> Is the second general-purpose source register, encoded in the "Rm" field.  
 <Ra> Is the third general-purpose source register holding the addend, encoded in the "Ra" field.

#### Operation for all encodings

```

1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 operand2 = if m_swap then ROR(R[m],16) else R[m];
4 product1 = SInt(R[n][15:0]) * SInt(operand2[15:0]);
5 product2 = SInt(R[n][31:16]) * SInt(operand2[31:16]);
6 result = product1 + product2 + SInt(R[a]);
7 R[d] = result[31:0];
8 if result != SInt(result[31:0]) then // Signed overflow
9 APSR.Q = '1';

```

## C2.4.157 SMLAL

Signed Multiply Accumulate Long. Signed Multiply Accumulate Long multiplies two signed 32-bit values to produce a 64-bit value, and accumulates this with a 64-bit value.

### T1

Armv8-M Main Extension only

|    |    |    |    |    |    |   |   |   |   |   |   |    |   |   |   |      |    |    |    |      |    |   |   |   |   |   |   |    |   |   |   |
|----|----|----|----|----|----|---|---|---|---|---|---|----|---|---|---|------|----|----|----|------|----|---|---|---|---|---|---|----|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3  | 2 | 1 | 0 | 15   | 14 | 13 | 12 | 11   | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3  | 2 | 1 | 0 |
| 1  | 1  | 1  | 1  | 1  | 0  | 1 | 1 | 1 | 1 | 0 | 0 | Rn |   |   |   | RdLo |    |    |    | RdHi |    |   |   | 0 | 0 | 0 | 0 | Rm |   |   |   |

### T1 variant

SMLAL{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>

### Decode for this encoding

```

1 if !HaveMainExt() then UNDEFINED;
2 dLo = UInt(RdLo); dHi = UInt(RdHi); n = UInt(Rn); m = UInt(Rm); setflags = FALSE;
3 if dLo IN {13,15} || dHi IN {13,15} || n IN {13,15} || m IN {13,15} then UNPREDICTABLE;
4 if dHi == dLo then UNPREDICTABLE;

```

### CONSTRAINED UNPREDICTABLE behavior

If `dHi == dLo`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The value in the destination register is UNKNOWN.

### Assembler symbols for all encodings

|        |                                                                                                                                                                                |
|--------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <c>    | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                                                                                                       |
| <q>    | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                                                                                                       |
| <RdLo> | Is the general-purpose source register holding the lower 32 bits of the addend, and the destination register for the lower 32 bits of the result, encoded in the "RdLo" field. |
| <RdHi> | Is the general-purpose source register holding the upper 32 bits of the addend, and the destination register for the upper 32 bits of the result, encoded in the "RdHi" field. |
| <Rn>   | Is the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.                                                                              |
| <Rm>   | Is the second general-purpose source register holding the multiplier, encoded in the "Rm" field.                                                                               |

### Operation for all encodings

```

1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 result = SInt(R[n]) * SInt(R[m]) + SInt(R[dHi]:R[dLo]);
4 R[dHi] = result[63:32];
5 R[dLo] = result[31:0];

```

### C2.4.158 SMLALBB, SMLALBT, SMLALTB, SMLALTT

Signed Multiply Accumulate Long (halfwords). Signed Multiply Accumulate Long (halfwords) multiplies two signed 16-bit values to produce a 32-bit value, and accumulates this with a 64-bit value. The multiply acts on two signed 16-bit quantities, taken from either the bottom or the top half of their respective source registers. The other halves of these source registers are ignored. The 32-bit product is sign-extended and accumulated with a 64-bit accumulate value.

Overflow is possible during this instruction, but only as a result of the 64-bit addition. This overflow is not detected if it occurs. Instead, the result wraps around modulo  $2^{64}$ .

#### T1

Armv8-M DSP Extension only

|    |    |    |    |    |    |   |   |   |   |   |   |    |   |   |   |      |    |    |    |      |    |   |   |   |   |   |   |    |   |   |   |
|----|----|----|----|----|----|---|---|---|---|---|---|----|---|---|---|------|----|----|----|------|----|---|---|---|---|---|---|----|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3  | 2 | 1 | 0 | 15   | 14 | 13 | 12 | 11   | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3  | 2 | 1 | 0 |
| 1  | 1  | 1  | 1  | 1  | 0  | 1 | 1 | 1 | 1 | 0 | 0 | Rn |   |   |   | RdLo |    |    |    | RdHi |    |   |   | 1 | 0 | N | M | Rm |   |   |   |

#### SMLALBB variant

Applies when **N == 0** && **M == 0**.

SMLALBB{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>

#### SMLALBT variant

Applies when **N == 0** && **M == 1**.

SMLALBT{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>

#### SMLALTB variant

Applies when **N == 1** && **M == 0**.

SMLALTB{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>

#### SMLALTT variant

Applies when **N == 1** && **M == 1**.

SMLALTT{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>

#### Decode for this encoding

```

1 if !HaveDSPExt() then UNDEFINED;
2 dLo = UInt(RdLo); dHi = UInt(RdHi); n = UInt(Rn); m = UInt(Rm);
3 n_high = (N == '1'); m_high = (M == '1');
4 if dLo IN {13,15} || dHi IN {13,15} || n IN {13,15} || m IN {13,15} then UNPREDICTABLE;
5 if dHi == dLo then UNPREDICTABLE;

```

#### CONSTRAINED UNPREDICTABLE behavior

If **dHi == dLo**, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The value in the destination register is UNKNOWN.

### Assembler symbols for all encodings

|        |                                                                                                                                                                                |
|--------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <c>    | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                                                                                                       |
| <q>    | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                                                                                                       |
| <RdLo> | Is the general-purpose source register holding the lower 32 bits of the addend, and the destination register for the lower 32 bits of the result, encoded in the "RdLo" field. |
| <RdHi> | Is the general-purpose source register holding the upper 32 bits of the addend, and the destination register for the upper 32 bits of the result, encoded in the "RdHi" field. |
| <Rn>   | Is the first general-purpose source register holding the multiplicand in the bottom or top half (selected by <x>), encoded in the "Rn" field.                                  |
| <Rm>   | Is the second general-purpose source register holding the multiplier in the bottom or top half (selected by <x>), encoded in the "Rm" field.                                   |

### Operation for all encodings

```
1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 operand1 = if n_high then R[n][31:16] else R[n][15:0];
4 operand2 = if m_high then R[m][31:16] else R[m][15:0];
5 result = SInt(operand1) * SInt(operand2) + SInt(R[dHi]:R[dLo]);
6 R[dHi] = result[63:32];
7 R[dLo] = result[31:0];
```

## C2.4.159 SMLALD, SMLALDX

Signed Multiply Accumulate Long Dual. Signed Multiply Accumulate Long Dual performs two signed 16-bit by 16-bit multiplications. It adds the products to a 64-bit accumulate operand.

Optionally, the instruction can exchange the halfwords of the second operand before performing the arithmetic.

Overflow is possible during this instruction, but only as a result of the 64-bit addition. This overflow is not detected if it occurs. Instead, the result wraps around modulo  $2^{64}$ .

### T1

Armv8-M DSP Extension only

|    |    |    |    |    |    |   |   |   |   |   |   |    |   |   |   |      |    |    |    |      |    |   |   |   |   |   |   |    |   |   |   |
|----|----|----|----|----|----|---|---|---|---|---|---|----|---|---|---|------|----|----|----|------|----|---|---|---|---|---|---|----|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3  | 2 | 1 | 0 | 15   | 14 | 13 | 12 | 11   | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3  | 2 | 1 | 0 |
| 1  | 1  | 1  | 1  | 1  | 0  | 1 | 1 | 1 | 1 | 0 | 0 | Rn |   |   |   | RdLo |    |    |    | RdHi |    |   |   | 1 | 1 | 0 | M | Rm |   |   |   |

#### SMLALD variant

Applies when **M == 0**.

SMLALD{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>

#### SMLALDX variant

Applies when **M == 1**.

SMLALDX{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>

#### Decode for this encoding

```

1 if !HaveDSPExt() then UNDEFINED;
2 dLo = UInt(RdLo); dHi = UInt(RdHi); n = UInt(Rn); m = UInt(Rm); m_swap = (M == '1');
3 if dLo IN {13,15} || dHi IN {13,15} || n IN {13,15} || m IN {13,15} then UNPREDICTABLE;
4 if dHi == dLo then UNPREDICTABLE;

```

#### CONSTRAINED UNPREDICTABLE behavior

If  $dHi == dLo$ , then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The value in the destination register is UNKNOWN.

#### Assembler symbols for all encodings

|        |                                                                                                                                                                                |
|--------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <c>    | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                                                                                                       |
| <q>    | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                                                                                                       |
| <RdLo> | Is the general-purpose source register holding the lower 32 bits of the addend, and the destination register for the lower 32 bits of the result, encoded in the "RdLo" field. |
| <RdHi> | Is the general-purpose source register holding the upper 32 bits of the addend, and the destination register for the upper 32 bits of the result, encoded in the "RdHi" field. |
| <Rn>   | Is the first general-purpose source register, encoded in the "Rn" field.                                                                                                       |
| <Rm>   | Is the second general-purpose source register, encoded in the "Rm" field.                                                                                                      |

#### Operation for all encodings

```

1 if ConditionPassed() then
2 EncodingSpecificOperations();

```



```
3 operand2 = if m_swap then ROR(R[m],16) else R[m];
4 product1 = SInt(R[n][15:0]) * SInt(operand2[15:0]);
5 product2 = SInt(R[n][31:16]) * SInt(operand2[31:16]);
6 result = product1 + product2 + SInt(R[dHi]:R[dLo]);
7 R[dHi] = result[63:32];
8 R[dLo] = result[31:0];
```

## C2.4.160 SMLAWB, SMLAWT

Signed Multiply Accumulate (word by halfword). Signed Multiply Accumulate (word by halfword) performs a signed multiply accumulate operation. The multiply acts on a signed 32-bit quantity and a signed 16-bit quantity. The signed 16-bit quantity is taken from either the bottom or the top half of its source register. The other half of the second source register is ignored. The top 32 bits of the 48-bit product are added to a 32-bit accumulate value and the result is written to the destination register. The bottom 16 bits of the 48-bit product are ignored.

If overflow occurs during the addition of the accumulate value, the instruction sets the Q flag in the APSR. No overflow can occur during the multiplication.

### T1

Armv8-M DSP Extension only

|    |    |    |    |    |    |   |   |   |   |   |   |    |   |   |   |            |    |    |    |    |    |   |   |   |   |   |   |    |   |   |   |
|----|----|----|----|----|----|---|---|---|---|---|---|----|---|---|---|------------|----|----|----|----|----|---|---|---|---|---|---|----|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3  | 2 | 1 | 0 | 15         | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3  | 2 | 1 | 0 |
| 1  | 1  | 1  | 1  | 1  | 0  | 1 | 1 | 0 | 0 | 1 | 1 | Rn |   |   |   | Ra != 1111 |    |    |    | Rd |    |   |   | 0 | 0 | 0 | M | Rm |   |   |   |

#### SMLAWB variant

Applies when **M == 0**.

SMLAWB{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>

#### SMLAWT variant

Applies when **M == 1**.

SMLAWT{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>

#### Decode for this encoding

```

1 if Ra == '1111' then SEE "SMULWB, SMULWT";
2 if !HaveDSPExt() then UNDEFINED;
3 d = UInt(Rd); n = UInt(Rn); m = UInt(Rm); a = UInt(Ra); m_high = (M == '1');
4 if d IN {13,15} || n IN {13,15} || m IN {13,15} || a == 13 then UNPREDICTABLE;

```

#### Assembler symbols for all encodings

|      |                                                                                                                                              |
|------|----------------------------------------------------------------------------------------------------------------------------------------------|
| <c>  | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                                                                     |
| <q>  | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                                                                     |
| <Rd> | Is the general-purpose destination register, encoded in the "Rd" field.                                                                      |
| <Rn> | Is the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.                                            |
| <Rm> | Is the second general-purpose source register holding the multiplier in the bottom or top half (selected by <y>), encoded in the "Rm" field. |
| <Ra> | Is the third general-purpose source register holding the addend, encoded in the "Ra" field.                                                  |

#### Operation for all encodings

```

1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 operand2 = if m_high then R[m][31:16] else R[m][15:0];
4 result = SInt(R[n]) * SInt(operand2) + (SInt(R[a]) << 16);
5 R[d] = result[47:16];
6 if (result >> 16) != SInt(R[d]) then // Signed overflow
7 APSR.Q = '1';

```

## C2.4.161 SMLSD, SMLSDX

Signed Multiply Subtract Dual. Signed Multiply Subtract Dual performs two signed 16-bit by 16-bit multiplications. It adds the difference of the products to a 32-bit accumulate operand.

Optionally, the instruction can exchange the halfwords of the second operand before performing the arithmetic.

This instruction sets the Q flag if the accumulate operation overflows. Overflow cannot occur during the multiplications or subtraction.

### T1

Armv8-M DSP Extension only

|    |    |    |    |    |    |   |   |   |   |   |   |    |   |   |   |            |    |    |    |    |    |   |   |   |   |   |   |    |   |   |   |
|----|----|----|----|----|----|---|---|---|---|---|---|----|---|---|---|------------|----|----|----|----|----|---|---|---|---|---|---|----|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3  | 2 | 1 | 0 | 15         | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3  | 2 | 1 | 0 |
| 1  | 1  | 1  | 1  | 1  | 0  | 1 | 1 | 0 | 1 | 0 | 0 | Rn |   |   |   | Ra != 1111 |    |    |    | Rd |    |   |   | 0 | 0 | 0 | M | Rm |   |   |   |

#### SMLSD variant

Applies when **M == 0**.

SMLSD{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>

#### SMLSDX variant

Applies when **M == 1**.

SMLSDX{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>

#### Decode for this encoding

```

1 if Ra == '1111' then SEE SMUSD;
2 if !HaveDSPEExt() then UNDEFINED;
3 d = UInt(Rd); n = UInt(Rn); m = UInt(Rm); a = UInt(Ra); m_swap = (M == '1');
4 if d IN {13,15} || n IN {13,15} || m IN {13,15} || a == 13 then UNPREDICTABLE;

```

#### Assembler symbols for all encodings

<c> See [C1.2.5 Standard assembler syntax fields](#) on page 309.  
 <q> See [C1.2.5 Standard assembler syntax fields](#) on page 309.  
 <Rd> Is the general-purpose destination register, encoded in the "Rd" field.  
 <Rn> Is the first general-purpose source register, encoded in the "Rn" field.  
 <Rm> Is the second general-purpose source register, encoded in the "Rm" field.  
 <Ra> Is the third general-purpose source register holding the addend, encoded in the "Ra" field.

#### Operation for all encodings

```

1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 operand2 = if m_swap then ROR(R[m],16) else R[m];
4 product1 = SInt(R[n][15:0]) * SInt(operand2[15:0]);
5 product2 = SInt(R[n][31:16]) * SInt(operand2[31:16]);
6 result = product1 - product2 + SInt(R[a]);
7 R[d] = result[31:0];
8 if result != SInt(result[31:0]) then // Signed overflow
9 APSR.Q = '1';

```

## C2.4.162 SMLS LD, SMLS LD X

Signed Multiply Subtract Long Dual. Signed Multiply Subtract Long Dual performs two signed 16-bit by 16-bit multiplications. It adds the difference of the products to a 64-bit accumulate operand.

Optionally, the instruction can exchange the halfwords of the second operand before performing the arithmetic.

Overflow is possible during this instruction, but only as a result of the 64-bit addition. This overflow is not detected if it occurs. Instead, the result wraps around modulo  $2^{64}$ .

### T1

Armv8-M DSP Extension only

|    |    |    |    |    |    |   |   |   |   |   |   |    |   |   |   |      |    |    |    |      |    |   |   |   |   |   |   |    |   |   |   |
|----|----|----|----|----|----|---|---|---|---|---|---|----|---|---|---|------|----|----|----|------|----|---|---|---|---|---|---|----|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3  | 2 | 1 | 0 | 15   | 14 | 13 | 12 | 11   | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3  | 2 | 1 | 0 |
| 1  | 1  | 1  | 1  | 1  | 0  | 1 | 1 | 1 | 1 | 0 | 1 | Rn |   |   |   | RdLo |    |    |    | RdHi |    |   |   | 1 | 1 | 0 | M | Rm |   |   |   |

### SMLS LD variant

Applies when **M == 0**.

SMLS LD{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>

### SMLS LD X variant

Applies when **M == 1**.

SMLS LD X{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>

### Decode for this encoding

```

1 if !HaveDSPExt() then UNDEFINED;
2 dLo = UInt(RdLo); dHi = UInt(RdHi); n = UInt(Rn); m = UInt(Rm); m_swap = (M == '1');
3 if dLo IN {13,15} || dHi IN {13,15} || n IN {13,15} || m IN {13,15} then UNPREDICTABLE;
4 if dHi == dLo then UNPREDICTABLE;

```

### CONSTRAINED UNPREDICTABLE behavior

If  $dHi == dLo$ , then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The value in the destination register is UNKNOWN.

### Assembler symbols for all encodings

|        |                                                                                                                                                                                |
|--------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <c>    | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                                                                                                       |
| <q>    | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                                                                                                       |
| <RdLo> | Is the general-purpose source register holding the lower 32 bits of the addend, and the destination register for the lower 32 bits of the result, encoded in the "RdLo" field. |
| <RdHi> | Is the general-purpose source register holding the upper 32 bits of the addend, and the destination register for the upper 32 bits of the result, encoded in the "RdHi" field. |
| <Rn>   | Is the first general-purpose source register, encoded in the "Rn" field.                                                                                                       |
| <Rm>   | Is the second general-purpose source register, encoded in the "Rm" field.                                                                                                      |

### Operation for all encodings

```

1 if ConditionPassed() then
2 EncodingSpecificOperations();

```

```
3 operand2 = if m_swap then ROR(R[m],16) else R[m];
4 product1 = SInt(R[n][15:0]) * SInt(operand2[15:0]);
5 product2 = SInt(R[n][31:16]) * SInt(operand2[31:16]);
6 result = product1 - product2 + SInt(R[dHi]:R[dLo]);
7 R[dHi] = result[63:32];
8 R[dLo] = result[31:0];
```

## C2.4.163 SMMLA, SMMLAR

Signed Most Significant Word Multiply Accumulate. Signed Most Significant Word Multiply Accumulate multiplies two signed 32-bit values, extracts the most significant 32 bits of the result, and adds an accumulate value.

Optionally, the instruction can specify that the result is rounded instead of being truncated. In this case, the constant  $0 \times 80000000$  is added to the product before the high word is extracted.

### T1

*Armv8-M DSP Extension only*

|    |    |    |    |    |    |   |   |   |   |   |   |    |   |   |   |            |    |    |    |    |    |   |   |       |   |   |   |    |   |   |   |
|----|----|----|----|----|----|---|---|---|---|---|---|----|---|---|---|------------|----|----|----|----|----|---|---|-------|---|---|---|----|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3  | 2 | 1 | 0 | 15         | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7     | 6 | 5 | 4 | 3  | 2 | 1 | 0 |
| 1  | 1  | 1  | 1  | 1  | 0  | 1 | 1 | 0 | 1 | 0 | 1 | Rn |   |   |   | Ra != 1111 |    |    |    | Rd |    |   |   | 0 0 0 |   |   | R | Rm |   |   |   |

#### SMMLA variant

Applies when **R == 0**.

SMMLA{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>

#### SMMLAR variant

Applies when **R == 1**.

SMMLAR{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>

#### Decode for this encoding

```

1 if Ra == '1111' then SEE SMMUL;
2 if !HaveDSPExt() then UNDEFINED;
3 d = UInt(Rd); n = UInt(Rn); m = UInt(Rm); a = UInt(Ra); round = (R == '1');
4 if d IN {13,15} || n IN {13,15} || m IN {13,15} || a == 13 then UNPREDICTABLE;
```

#### Assembler symbols for all encodings

|      |                                                                                                   |
|------|---------------------------------------------------------------------------------------------------|
| <c>  | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                          |
| <q>  | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                          |
| <Rd> | Is the general-purpose destination register, encoded in the "Rd" field.                           |
| <Rn> | Is the first general-purpose source register holding the multiplicand, encoded in the "Rn" field. |
| <Rm> | Is the second general-purpose source register holding the multiplier, encoded in the "Rm" field.  |
| <Ra> | Is the third general-purpose source register holding the addend, encoded in the "Ra" field.       |

#### Operation for all encodings

```

1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 result = (SInt(R[a]) << 32) + SInt(R[n]) * SInt(R[m]);
4 if round then result = result + 0x80000000;
5 R[d] = result[63:32];
```

## C2.4.164 SMMLS, SMMLSR

Signed Most Significant Word Multiply Subtract. Signed Most Significant Word Multiply Subtract multiplies two signed 32-bit values, subtracts the result from a 32-bit accumulate value that is shifted left by 32 bits, and extracts the most significant 32 bits of the result of that subtraction.

Optionally, the instruction can specify that the result of the instruction is rounded instead of being truncated. In this case, the constant 0x80000000 is added to the result of the subtraction before the high word is extracted.

### T1

*Armv8-M DSP Extension only*

|    |    |    |    |    |    |   |   |   |   |   |   |    |   |   |   |    |    |    |    |    |    |   |   |   |   |   |   |    |   |   |   |
|----|----|----|----|----|----|---|---|---|---|---|---|----|---|---|---|----|----|----|----|----|----|---|---|---|---|---|---|----|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3  | 2 | 1 | 0 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3  | 2 | 1 | 0 |
| 1  | 1  | 1  | 1  | 1  | 0  | 1 | 1 | 0 | 1 | 1 | 0 | Rn |   |   |   | Ra |    |    |    | Rd |    |   |   | 0 | 0 | 0 | R | Rm |   |   |   |

#### SMMLS variant

Applies when **R == 0**.

SMMLS{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>

#### SMMLSR variant

Applies when **R == 1**.

SMMLSR{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>

#### Decode for this encoding

```

1 if !HaveDSPExt() then UNDEFINED;
2 d = UInt(Rd); n = UInt(Rn); m = UInt(Rm); a = UInt(Ra); round = (R == '1');
3 if d IN {13,15} || n IN {13,15} || m IN {13,15} || a IN {13,15} then UNPREDICTABLE;

```

#### Assembler symbols for all encodings

|      |                                                                                                   |
|------|---------------------------------------------------------------------------------------------------|
| <c>  | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                          |
| <q>  | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                          |
| <Rd> | Is the general-purpose destination register, encoded in the "Rd" field.                           |
| <Rn> | Is the first general-purpose source register holding the multiplicand, encoded in the "Rn" field. |
| <Rm> | Is the second general-purpose source register holding the multiplier, encoded in the "Rm" field.  |
| <Ra> | Is the third general-purpose source register holding the addend, encoded in the "Ra" field.       |

#### Operation for all encodings

```

1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 result = (SInt(R[a]) << 32) - SInt(R[n]) * SInt(R[m]);
4 if round then result = result + 0x80000000;
5 R[d] = result[63:32];

```

## C2.4.165 SMMUL, SMMULR

Signed Most Significant Word Multiply. Signed Most Significant Word Multiply multiplies two signed 32-bit values, extracts the most significant 32 bits of the result, and writes those bits to the destination register.

Optionally, the instruction can specify that the result is rounded instead of being truncated. In this case, the constant 0x80000000 is added to the product before the high word is extracted.

### T1

Armv8-M DSP Extension only

|    |    |    |    |    |    |   |   |   |   |   |   |    |   |   |   |    |    |    |    |    |    |   |   |   |   |   |   |    |   |   |   |
|----|----|----|----|----|----|---|---|---|---|---|---|----|---|---|---|----|----|----|----|----|----|---|---|---|---|---|---|----|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3  | 2 | 1 | 0 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3  | 2 | 1 | 0 |
| 1  | 1  | 1  | 1  | 1  | 0  | 1 | 1 | 0 | 1 | 0 | 1 | Rn |   |   |   | 1  | 1  | 1  | 1  | Rd |    |   |   | 0 | 0 | 0 | R | Rm |   |   |   |

#### SMMUL variant

Applies when R == 0.

SMMUL{<c>}{<q>} {<Rd>}, <Rn>, <Rm>

#### SMMULR variant

Applies when R == 1.

SMMULR{<c>}{<q>} {<Rd>}, <Rn>, <Rm>

#### Decode for this encoding

```

1 if !HaveDSPExt() then UNDEFINED;
2 d = UInt(Rd); n = UInt(Rn); m = UInt(Rm); round = (R == '1');
3 if d IN {13,15} || n IN {13,15} || m IN {13,15} then UNPREDICTABLE;

```

#### Assembler symbols for all encodings

<c> See [C1.2.5 Standard assembler syntax fields](#) on page 309.  
 <q> See [C1.2.5 Standard assembler syntax fields](#) on page 309.  
 <Rd> Is the general-purpose destination register, encoded in the "Rd" field.  
 <Rn> Is the first general-purpose source register, encoded in the "Rn" field.  
 <Rm> Is the second general-purpose source register, encoded in the "Rm" field.

#### Operation for all encodings

```

1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 result = SInt(R[n]) * SInt(R[m]);
4 if round then result = result + 0x80000000;
5 R[d] = result[63:32];

```



## C2.4.166 SMUAD, SMUADX

Signed Dual Multiply Add. Signed Dual Multiply Add performs two signed 16-bit by 16-bit multiplications. It adds the products together, and writes the result to the destination register.

Optionally, the instruction can exchange the halfwords of the second operand before performing the arithmetic.

This instruction sets the Q flag if the addition overflows. The multiplications cannot overflow.

### T1

Armv8-M DSP Extension only

|    |    |    |    |    |    |   |   |   |   |   |   |    |   |   |   |    |    |    |    |    |    |   |   |   |   |   |   |    |   |   |   |
|----|----|----|----|----|----|---|---|---|---|---|---|----|---|---|---|----|----|----|----|----|----|---|---|---|---|---|---|----|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3  | 2 | 1 | 0 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3  | 2 | 1 | 0 |
| 1  | 1  | 1  | 1  | 1  | 0  | 1 | 1 | 0 | 0 | 1 | 0 | Rn |   |   |   | 1  | 1  | 1  | 1  | Rd |    |   |   | 0 | 0 | 0 | M | Rm |   |   |   |

#### SMUAD variant

Applies when **M == 0**.

SMUAD{<c>}{<q>} {<Rd>}, <Rn>, <Rm>

#### SMUADX variant

Applies when **M == 1**.

SMUADX{<c>}{<q>} {<Rd>}, <Rn>, <Rm>

#### Decode for this encoding

```

1 if !HaveDSPExt() then UNDEFINED;
2 d = UInt(Rd); n = UInt(Rn); m = UInt(Rm); m_swap = (M == '1');
3 if d IN {13,15} || n IN {13,15} || m IN {13,15} then UNPREDICTABLE;

```

#### Assembler symbols for all encodings

<c> See [C1.2.5 Standard assembler syntax fields](#) on page 309.  
 <q> See [C1.2.5 Standard assembler syntax fields](#) on page 309.  
 <Rd> Is the general-purpose destination register, encoded in the "Rd" field.  
 <Rn> Is the first general-purpose source register, encoded in the "Rn" field.  
 <Rm> Is the second general-purpose source register, encoded in the "Rm" field.

#### Operation for all encodings

```

1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 operand2 = if m_swap then ROR(R[m],16) else R[m];
4 product1 = SInt(R[n][15:0]) * SInt(operand2[15:0]);
5 product2 = SInt(R[n][31:16]) * SInt(operand2[31:16]);
6 result = product1 + product2;
7 R[d] = result[31:0];
8 if result != SInt(result[31:0]) then // Signed overflow
9 APSR.Q = '1';

```

## C2.4.167 SMULBB, SMULBT, SMULTB, SMULTT

Signed Multiply (halfwords). Signed Multiply (halfwords) multiplies two signed 16-bit quantities, taken from either the bottom or the top half of their respective source registers. The other halves of these source registers are ignored. The 32-bit product is written to the destination register. No overflow is possible during this instruction.

### T1

Armv8-M DSP Extension only

|    |    |    |    |    |    |   |   |   |   |   |   |    |   |   |   |    |    |    |    |    |    |   |   |   |   |   |   |    |   |   |   |
|----|----|----|----|----|----|---|---|---|---|---|---|----|---|---|---|----|----|----|----|----|----|---|---|---|---|---|---|----|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3  | 2 | 1 | 0 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3  | 2 | 1 | 0 |
| 1  | 1  | 1  | 1  | 1  | 0  | 1 | 1 | 0 | 0 | 0 | 1 | Rn |   |   |   | 1  | 1  | 1  | 1  | Rd |    |   |   | 0 | 0 | N | M | Rm |   |   |   |

#### SMULBB variant

Applies when **N == 0** && **M == 0**.

SMULBB{<c>}{<q>} {<Rd>}, {<Rn>}, <Rm>

#### SMULBT variant

Applies when **N == 0** && **M == 1**.

SMULBT{<c>}{<q>} {<Rd>}, {<Rn>}, <Rm>

#### SMULTB variant

Applies when **N == 1** && **M == 0**.

SMULTB{<c>}{<q>} {<Rd>}, {<Rn>}, <Rm>

#### SMULTT variant

Applies when **N == 1** && **M == 1**.

SMULTT{<c>}{<q>} {<Rd>}, {<Rn>}, <Rm>

### Decode for this encoding

```

1 if !HaveDSPExt() then UNDEFINED;
2 d = UInt(Rd); n = UInt(Rn); m = UInt(Rm);
3 n_high = (N == '1'); m_high = (M == '1');
4 if d IN {13,15} || n IN {13,15} || m IN {13,15} then UNPREDICTABLE;

```

### Assembler symbols for all encodings

|      |                                                                                                                                               |
|------|-----------------------------------------------------------------------------------------------------------------------------------------------|
| <c>  | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                                                                      |
| <q>  | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                                                                      |
| <Rd> | Is the general-purpose destination register, encoded in the "Rd" field.                                                                       |
| <Rn> | Is the first general-purpose source register holding the multiplicand in the bottom or top half (selected by <x>), encoded in the "Rn" field. |
| <Rm> | Is the second general-purpose source register holding the multiplier in the bottom or top half (selected by <y>), encoded in the "Rm" field.  |

### Operation for all encodings

```

1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 operand1 = if n_high then R[n][31:16] else R[n][15:0];
4 operand2 = if m_high then R[m][31:16] else R[m][15:0];

```

Chapter C2. Instruction Specification

C2.4. Alphabetical list of instructions

```
5 result = SInt(operand1) * SInt(operand2);
6 R[d] = result[31:0];
7 // Signed overflow cannot occur
```

## C2.4.168 SMULL

Signed Multiply Long. Signed Multiply Long multiplies two 32-bit signed values to produce a 64-bit result.

### T1

Armv8-M Main Extension only

|    |    |    |    |    |    |   |   |   |   |   |   |    |   |   |      |    |    |      |    |    |    |   |   |   |    |   |   |   |   |   |   |
|----|----|----|----|----|----|---|---|---|---|---|---|----|---|---|------|----|----|------|----|----|----|---|---|---|----|---|---|---|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3  | 2 | 1 | 0    | 15 | 14 | 13   | 12 | 11 | 10 | 9 | 8 | 7 | 6  | 5 | 4 | 3 | 2 | 1 | 0 |
| 1  | 1  | 1  | 1  | 1  | 0  | 1 | 1 | 1 | 0 | 0 | 0 | Rn |   |   | RdLo |    |    | RdHi |    |    | 0  | 0 | 0 | 0 | Rm |   |   |   |   |   |   |

### T1 variant

SMULL{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>

### Decode for this encoding

```

1 if !HaveMainExt() then UNDEFINED;
2 dLo = UInt(RdLo); dHi = UInt(RdHi); n = UInt(Rn); m = UInt(Rm); setflags = FALSE;
3 if dLo IN {13,15} || dHi IN {13,15} || n IN {13,15} || m IN {13,15} then UNPREDICTABLE;
4 if dHi == dLo then UNPREDICTABLE;

```

### CONSTRAINED UNPREDICTABLE behavior

If `dHi == dLo`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The value in the destination register is UNKNOWN.

### Assembler symbols for all encodings

|        |                                                                                                               |
|--------|---------------------------------------------------------------------------------------------------------------|
| <c>    | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                                      |
| <q>    | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                                      |
| <RdLo> | Is the general-purpose destination register for the lower 32 bits of the result, encoded in the "RdLo" field. |
| <RdHi> | Is the general-purpose destination register for the upper 32 bits of the result, encoded in the "RdHi" field. |
| <Rn>   | Is the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.             |
| <Rm>   | Is the second general-purpose source register holding the multiplier, encoded in the "Rm" field.              |

### Operation for all encodings

```

1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 result = SInt(R[n]) * SInt(R[m]);
4 R[dHi] = result[63:32];
5 R[dLo] = result[31:0];

```

## C2.4.169 SMULWB, SMULWT

Signed Multiply (word by halfword). Signed Multiply (word by halfword) multiplies a signed 32-bit quantity and a signed 16-bit quantity. The signed 16-bit quantity is taken from either the bottom or the top half of its source register. The other half of the second source register is ignored. The top 32 bits of the 48-bit product are written to the destination register. The bottom 16 bits of the 48-bit product are ignored. No overflow is possible during this instruction.

### T1

Armv8-M DSP Extension only

|    |    |    |    |    |    |   |   |   |   |   |   |    |   |   |   |    |    |    |    |    |    |   |   |   |   |   |   |    |   |   |   |
|----|----|----|----|----|----|---|---|---|---|---|---|----|---|---|---|----|----|----|----|----|----|---|---|---|---|---|---|----|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3  | 2 | 1 | 0 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3  | 2 | 1 | 0 |
| 1  | 1  | 1  | 1  | 1  | 0  | 1 | 1 | 0 | 0 | 1 | 1 | Rn |   |   |   | 1  | 1  | 1  | 1  | Rd |    |   |   | 0 | 0 | 0 | M | Rm |   |   |   |

### SMULWB variant

Applies when **M == 0**.

SMULWB{<c>}{<q>} {<Rd>}, {<Rn>}, <Rm>

### SMULWT variant

Applies when **M == 1**.

SMULWT{<c>}{<q>} {<Rd>}, {<Rn>}, <Rm>

### Decode for this encoding

```

1 if !HaveDSPExt() then UNDEFINED;
2 d = UInt(Rd); n = UInt(Rn); m = UInt(Rm); m_high = (M == '1');
3 if d IN {13,15} || n IN {13,15} || m IN {13,15} then UNPREDICTABLE;

```

### Assembler symbols for all encodings

|      |                                                                                                                                              |
|------|----------------------------------------------------------------------------------------------------------------------------------------------|
| <c>  | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                                                                     |
| <q>  | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                                                                     |
| <Rd> | Is the general-purpose destination register, encoded in the "Rd" field.                                                                      |
| <Rn> | Is the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.                                            |
| <Rm> | Is the second general-purpose source register holding the multiplier in the bottom or top half (selected by <y>), encoded in the "Rm" field. |

### Operation for all encodings

```

1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 operand2 = if m_high then R[m][31:16] else R[m][15:0];
4 product = SInt(R[n]) * SInt(operand2);
5 R[d] = product[47:16];
6 // Signed overflow cannot occur

```

## C2.4.170 SMUSD, SMUSDX

Signed Dual Multiply Subtract. Signed Dual Multiply Subtract performs two signed 16-bit by 16-bit multiplications. It subtracts one of the products from the other, and writes the result to the destination register.

Optionally, the instruction can exchange the halfwords of the second operand before performing the arithmetic.

Overflow cannot occur.

### T1

Armv8-M DSP Extension only

|    |    |    |    |    |    |   |   |   |   |   |   |    |   |   |   |    |    |    |    |    |    |   |   |   |   |   |   |    |   |   |   |
|----|----|----|----|----|----|---|---|---|---|---|---|----|---|---|---|----|----|----|----|----|----|---|---|---|---|---|---|----|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3  | 2 | 1 | 0 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3  | 2 | 1 | 0 |
| 1  | 1  | 1  | 1  | 1  | 0  | 1 | 1 | 0 | 1 | 0 | 0 | Rn |   |   |   | 1  | 1  | 1  | 1  | Rd |    |   |   | 0 | 0 | 0 | M | Rm |   |   |   |

#### SMUSD variant

Applies when **M == 0**.

SMUSD{<c>}{<q>} {<Rd>}, <Rn>, <Rm>

#### SMUSDX variant

Applies when **M == 1**.

SMUSDX{<c>}{<q>} {<Rd>}, <Rn>, <Rm>

#### Decode for this encoding

```

1 if !HaveDSPExt() then UNDEFINED;
2 d = UInt(Rd); n = UInt(Rn); m = UInt(Rm); m_swap = (M == '1');
3 if d IN {13,15} || n IN {13,15} || m IN {13,15} then UNPREDICTABLE;

```

#### Assembler symbols for all encodings

<c> See [C1.2.5 Standard assembler syntax fields](#) on page 309.  
 <q> See [C1.2.5 Standard assembler syntax fields](#) on page 309.  
 <Rd> Is the general-purpose destination register, encoded in the "Rd" field.  
 <Rn> Is the first general-purpose source register, encoded in the "Rn" field.  
 <Rm> Is the second general-purpose source register, encoded in the "Rm" field.

#### Operation for all encodings

```

1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 operand2 = if m_swap then ROR(R[m],16) else R[m];
4 product1 = SInt(R[n][15:0]) * SInt(operand2[15:0]);
5 product2 = SInt(R[n][31:16]) * SInt(operand2[31:16]);
6 result = product1 - product2;
7 R[d] = result[31:0];
8 // Signed overflow cannot occur

```

## C2.4.171 SSAT

Signed Saturate. Signed Saturate saturates an optionally-shifted signed value to a selectable signed range.

The APSR.Q flag is set to 1 if the operation saturates.

### T1

Armv8-M Main Extension only

|    |    |    |    |    |     |   |   |   |   |    |   |    |   |   |   |    |      |    |    |    |    |   |   |   |      |     |         |   |   |   |   |
|----|----|----|----|----|-----|---|---|---|---|----|---|----|---|---|---|----|------|----|----|----|----|---|---|---|------|-----|---------|---|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10  | 9 | 8 | 7 | 6 | 5  | 4 | 3  | 2 | 1 | 0 | 15 | 14   | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6    | 5   | 4       | 3 | 2 | 1 | 0 |
| 1  | 1  | 1  | 1  | 0  | (0) | 1 | 1 | 0 | 0 | sh | 0 | Rn |   |   |   | 0  | imm3 |    |    |    | Rd |   |   |   | imm2 | (0) | sat_imm |   |   |   |   |

#### Arithmetic shift right variant

Applies when `sh == 1 && !(imm3 == 000 && imm2 == 00)`.

SSAT{<c>}{<q>} <Rd>, #<imm>, <Rn>, ASR #<amount>

#### Logical shift left variant

Applies when `sh == 0`.

SSAT{<c>}{<q>} <Rd>, #<imm>, <Rn> {, LSL #<amount>}

#### Decode for this encoding

```

1 if sh == '1' && (imm3:imm2) == '0000' then
2 if HaveDSPExt() then
3 SEE SSAT16;
4 else
5 UNDEFINED;
6 if !HaveMainExt() then UNDEFINED;
7 d = UInt(Rd); n = UInt(Rn); saturate_to = UInt(sat_imm)+1;
8 (shift_t, shift_n) = DecodeImmShift(sh:'0', imm3:imm2);
9 if d IN {13,15} || n IN {13,15} then UNPREDICTABLE;
```

#### Assembler symbols for all encodings

|          |                                                                                                                                                                                                                                                                                  |
|----------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <c>      | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                                                                                                                                                                                                         |
| <q>      | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                                                                                                                                                                                                         |
| <Rd>     | Is the general-purpose destination register, encoded in the "Rd" field.                                                                                                                                                                                                          |
| <imm>    | Is the bit position for saturation, in the range 1 to 32, encoded in the "sat_imm" field as <imm>-1.                                                                                                                                                                             |
| <Rn>     | Is the general-purpose source register, encoded in the "Rn" field.                                                                                                                                                                                                               |
| <amount> | For the arithmetic shift right variant: is the shift amount, in the range 1 to 31 encoded in the "imm3:imm2" field as <amount>.<br>For the logical shift left variant: is the optional shift amount, in the range 0 to 31, defaulting to 0 and encoded in the "imm3:imm2" field. |

#### Operation for all encodings

```

1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 operand = Shift(R[n], shift_t, shift_n, APSR.C); // APSR.C ignored
4 (result, sat) = SignedSatQ(SInt(operand), saturate_to);
5 R[d] = SignExtend(result, 32);
6 if sat then
7 APSR.Q = '1';
```

## C2.4.172 SSAT16

Signed Saturate 16. Signed Saturate 16 saturates two signed 16-bit values to a selected signed range.

The APSR.Q flag is set to 1 if the operation saturates.

### T1

*Armv8-M DSP Extension only*

|    |    |    |    |    |     |   |   |   |   |   |   |    |   |   |   |    |    |    |    |    |    |   |   |   |   |     |     |         |   |   |   |
|----|----|----|----|----|-----|---|---|---|---|---|---|----|---|---|---|----|----|----|----|----|----|---|---|---|---|-----|-----|---------|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10  | 9 | 8 | 7 | 6 | 5 | 4 | 3  | 2 | 1 | 0 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5   | 4   | 3       | 2 | 1 | 0 |
| 1  | 1  | 1  | 1  | 0  | (0) | 1 | 1 | 0 | 0 | 1 | 0 | Rn |   |   |   | 0  | 0  | 0  | 0  | Rd |    |   |   | 0 | 0 | (0) | (0) | sat_imm |   |   |   |

### T1 variant

SSAT16{<c>}{<q>} <Rd>, #<imm>, <Rn>

### Decode for this encoding

```
1 if !HaveDSPExt() then UNDEFINED;
2 d = UInt(Rd); n = UInt(Rn); saturate_to = UInt(sat_imm)+1;
3 if d IN {13,15} || n IN {13,15} then UNPREDICTABLE;
```

### Assembler symbols for all encodings

<c> See [C1.2.5 Standard assembler syntax fields](#) on page 309.  
 <q> See [C1.2.5 Standard assembler syntax fields](#) on page 309.  
 <Rd> Is the general-purpose destination register, encoded in the "Rd" field.  
 <imm> Is the bit position for saturation, in the range 1 to 16, encoded in the "sat\_imm" field as <imm>-1.  
 <Rn> Is the general-purpose source register, encoded in the "Rn" field.

### Operation for all encodings

```
1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 (result1, sat1) = SignedSatQ(SInt(R[n][15:0]), saturate_to);
4 (result2, sat2) = SignedSatQ(SInt(R[n][31:16]), saturate_to);
5 bits(32) result;
6 result[15:0] = SignExtend(result1, 16);
7 result[31:16] = SignExtend(result2, 16);
8 R[d] = result;
9 if sat1 || sat2 then
10 APSR.Q = '1';
```



### C2.4.173 SSAX

Signed Subtract and Add with Exchange. Signed Subtract and Add with Exchange exchanges the two halfwords of the second operand, performs one 16-bit integer subtraction and one 16-bit addition, and writes the results to the destination register. It sets the APSR.GE bits according to the results.

#### T1

Armv8-M DSP Extension only

|    |    |    |    |    |    |   |   |   |   |   |   |    |   |   |   |    |    |    |    |    |    |   |   |   |   |   |   |    |   |   |   |
|----|----|----|----|----|----|---|---|---|---|---|---|----|---|---|---|----|----|----|----|----|----|---|---|---|---|---|---|----|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3  | 2 | 1 | 0 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3  | 2 | 1 | 0 |
| 1  | 1  | 1  | 1  | 1  | 0  | 1 | 0 | 1 | 1 | 1 | 0 | Rn |   |   |   | 1  | 1  | 1  | 1  | Rd |    |   |   | 0 | 0 | 0 | 0 | Rm |   |   |   |

#### T1 variant

SSAX{<c>}{<q>} {<Rd>}, {<Rn>}, <Rm>

#### Decode for this encoding

```

1 if !HaveDSPExt() then UNDEFINED;
2 d = UInt(Rd); n = UInt(Rn); m = UInt(Rm);
3 if d IN {13,15} || n IN {13,15} || m IN {13,15} then UNPREDICTABLE;

```

#### Assembler symbols for all encodings

<c> See [C1.2.5 Standard assembler syntax fields](#) on page 309.  
 <q> See [C1.2.5 Standard assembler syntax fields](#) on page 309.  
 <Rd> Is the general-purpose destination register, encoded in the "Rd" field.  
 <Rn> Is the first general-purpose source register, encoded in the "Rn" field.  
 <Rm> Is the second general-purpose source register, encoded in the "Rm" field.

#### Operation for all encodings

```

1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 sum = SInt(R[n][15:0]) + SInt(R[m][31:16]);
4 diff = SInt(R[n][31:16]) - SInt(R[m][15:0]);
5 R[d] = diff[15:0] : sum[15:0];
6 APSR.GE[1:0] = if sum >= 0 then '11' else '00';
7 APSR.GE[3:2] = if diff >= 0 then '11' else '00';

```

### C2.4.174 SSBB

Speculative Store Bypass Barrier. Speculative Store Bypass Barrier is a memory barrier which prevents speculative loads from bypassing earlier stores to the same address under certain conditions.

The semantics of the Speculative Store Bypass Barrier are:

- When a load to a location appears in program order after the SSBB, then the load does not speculatively read an entry earlier in the coherence order for that location than the entry generated by the latest store satisfying all of the following conditions:
  - The store is to the same location as the load.
  - The store appears in program order before the SSBB.
- When a load to a location appears in program order before the SSBB, then the load does not speculatively read data from any store satisfying all of the following conditions:
  - The store is to the same location as the load.
  - The store appears in program order after the SSBB.

#### T1

*Armv8-M Main Extension only*

|    |    |    |    |    |    |   |   |   |   |   |   |     |     |     |     |    |    |     |    |     |     |     |     |   |   |   |   |   |   |   |   |
|----|----|----|----|----|----|---|---|---|---|---|---|-----|-----|-----|-----|----|----|-----|----|-----|-----|-----|-----|---|---|---|---|---|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3   | 2   | 1   | 0   | 15 | 14 | 13  | 12 | 11  | 10  | 9   | 8   | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 1  | 1  | 1  | 1  | 0  | 0  | 1 | 1 | 1 | 0 | 1 | 1 | (1) | (1) | (1) | (1) | 1  | 0  | (0) | 0  | (1) | (1) | (1) | (1) | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |

#### T1 variant

SSBB { <q> }

#### Decode for this encoding

```
1 if !HaveMainExt() then UNDEFINED;
2 if InITBlock() then UNPREDICTABLE;
```

#### Assembler symbols for all encodings

<q> See [C1.2.5 Standard assembler syntax fields](#) on page 309.

#### Operation for all encodings

```
1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 SpeculativeSynchronizationBarrier();
```

### C2.4.175 SSUB16

Signed Subtract 16. Signed Subtract 16 performs two 16-bit signed integer subtractions, and writes the results to the destination register. It sets the APSR.GE bits according to the results of the subtractions.

#### T1

Armv8-M DSP Extension only

|    |    |    |    |    |    |   |   |   |   |   |   |    |   |   |   |    |    |    |    |    |    |   |   |   |   |   |   |    |   |   |   |
|----|----|----|----|----|----|---|---|---|---|---|---|----|---|---|---|----|----|----|----|----|----|---|---|---|---|---|---|----|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3  | 2 | 1 | 0 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3  | 2 | 1 | 0 |
| 1  | 1  | 1  | 1  | 1  | 0  | 1 | 0 | 1 | 1 | 0 | 1 | Rn |   |   |   | 1  | 1  | 1  | 1  | Rd |    |   |   | 0 | 0 | 0 | 0 | Rm |   |   |   |

#### T1 variant

SSUB16{<c>}{<q>} {<Rd>}, {<Rn>}, <Rm>

#### Decode for this encoding

```

1 if !HaveDSPExt() then UNDEFINED;
2 d = UInt(Rd); n = UInt(Rn); m = UInt(Rm);
3 if d IN {13,15} || n IN {13,15} || m IN {13,15} then UNPREDICTABLE;

```

#### Assembler symbols for all encodings

<c> See [C1.2.5 Standard assembler syntax fields](#) on page 309.  
 <q> See [C1.2.5 Standard assembler syntax fields](#) on page 309.  
 <Rd> Is the general-purpose destination register, encoded in the "Rd" field.  
 <Rn> Is the first general-purpose source register, encoded in the "Rn" field.  
 <Rm> Is the second general-purpose source register, encoded in the "Rm" field.

#### Operation for all encodings

```

1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 diff1 = SInt(R[n][15:0]) - SInt(R[m][15:0]);
4 diff2 = SInt(R[n][31:16]) - SInt(R[m][31:16]);
5 R[d] = diff2[15:0] : diff1[15:0];
6 APSR.GE[1:0] = if diff1 >= 0 then '11' else '00';
7 APSR.GE[3:2] = if diff2 >= 0 then '11' else '00';

```

## C2.4.176 SSUB8

Signed Subtract 8. Signed Subtract 8 performs four 8-bit signed integer subtractions, and writes the results to the destination register. It sets the APSR.GE bits according to the results of the subtractions.

### T1

*Armv8-M DSP Extension only*

|    |    |    |    |    |    |   |   |   |   |   |   |    |   |   |   |    |    |    |    |    |    |   |   |   |   |   |   |    |   |   |   |
|----|----|----|----|----|----|---|---|---|---|---|---|----|---|---|---|----|----|----|----|----|----|---|---|---|---|---|---|----|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3  | 2 | 1 | 0 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3  | 2 | 1 | 0 |
| 1  | 1  | 1  | 1  | 1  | 0  | 1 | 0 | 1 | 1 | 0 | 0 | Rn |   |   |   | 1  | 1  | 1  | 1  | Rd |    |   |   | 0 | 0 | 0 | 0 | Rm |   |   |   |

### T1 variant

SSUB8 {<c>} {<q>} {<Rd>}, {<Rn>}, {<Rm>}

### Decode for this encoding

```

1 if !HaveDSPExt() then UNDEFINED;
2 d = UInt(Rd); n = UInt(Rn); m = UInt(Rm);
3 if d IN {13,15} || n IN {13,15} || m IN {13,15} then UNPREDICTABLE;

```

### Assembler symbols for all encodings

<c> See [C1.2.5 Standard assembler syntax fields](#) on page 309.  
 <q> See [C1.2.5 Standard assembler syntax fields](#) on page 309.  
 <Rd> Is the general-purpose destination register, encoded in the "Rd" field.  
 <Rn> Is the first general-purpose source register, encoded in the "Rn" field.  
 <Rm> Is the second general-purpose source register, encoded in the "Rm" field.

### Operation for all encodings

```

1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 diff1 = SInt(R[n][7:0]) - SInt(R[m][7:0]);
4 diff2 = SInt(R[n][15:8]) - SInt(R[m][15:8]);
5 diff3 = SInt(R[n][23:16]) - SInt(R[m][23:16]);
6 diff4 = SInt(R[n][31:24]) - SInt(R[m][31:24]);
7 R[d] = diff4[7:0] : diff3[7:0] : diff2[7:0] : diff1[7:0];
8 APSR.GE[0] = if diff1 >= 0 then '1' else '0';
9 APSR.GE[1] = if diff2 >= 0 then '1' else '0';
10 APSR.GE[2] = if diff3 >= 0 then '1' else '0';
11 APSR.GE[3] = if diff4 >= 0 then '1' else '0';

```

### C2.4.177 STC, STC2

Store Coprocessor. Store Coprocessor stores data from a coprocessor to a sequence of consecutive memory addresses.

If no coprocessor can execute the instruction, a UsageFault exception is generated.

Arm reserves coprocessors CP8 to CP15, and this manual defines the valid instructions when coproc is in this range.

#### T1

*Armv8-M Main Extension only*

|    |    |    |    |    |    |   |   |   |   |   |   |    |   |   |   |     |    |    |    |        |    |   |   |      |   |   |   |   |   |   |   |
|----|----|----|----|----|----|---|---|---|---|---|---|----|---|---|---|-----|----|----|----|--------|----|---|---|------|---|---|---|---|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3  | 2 | 1 | 0 | 15  | 14 | 13 | 12 | 11     | 10 | 9 | 8 | 7    | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 1  | 1  | 1  | 0  | 1  | 1  | 0 | P | U | D | W | 0 | Rn |   |   |   | CRd |    |    |    | coproc |    |   |   | imm8 |   |   |   |   |   |   |   |

#### Offset variant

Applies when **P == 1 && W == 0**.

STC{L}{<c>}{<q>} <coproc>, <CRd>, [<Rn>{, #<+/-><imm>}]

#### Post-indexed variant

Applies when **P == 0 && W == 1**.

STC{L}{<c>}{<q>} <coproc>, <CRd>, [<Rn>], #<+/-><imm>

#### Pre-indexed variant

Applies when **P == 1 && W == 1**.

STC{L}{<c>}{<q>} <coproc>, <CRd>, [<Rn>, #<+/-><imm>]!

#### Unindexed variant

Applies when **P == 0 && U == 1 && W == 0**.

STC{L}{<c>}{<q>} <coproc>, <CRd>, [<Rn>], <option>

#### Decode for this encoding

```

1 if P == '0' && U == '0' && D == '1' && W == '0' then SEE "MCRR, MCRR2";
2 if coproc IN '101x' then SEE "Floating-point";
3 if P == '0' && U == '0' && D == '0' && W == '0' then UNDEFINED;
4 if !HaveMainExt() then UNDEFINED;
5 n = UInt(Rn); cp = UInt(coproc); imm32 = ZeroExtend(imm8:'00', 32);
6 index = (P == '1'); add = (U == '1'); wback = (W == '1');
7 if n == 15 then UNPREDICTABLE;

```

#### T2

*Armv8-M Main Extension only*

|    |    |    |    |    |    |   |   |   |   |   |   |    |   |   |   |     |    |    |    |        |    |   |   |      |   |   |   |   |   |   |   |
|----|----|----|----|----|----|---|---|---|---|---|---|----|---|---|---|-----|----|----|----|--------|----|---|---|------|---|---|---|---|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3  | 2 | 1 | 0 | 15  | 14 | 13 | 12 | 11     | 10 | 9 | 8 | 7    | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 1  | 1  | 1  | 1  | 1  | 1  | 0 | P | U | D | W | 0 | Rn |   |   |   | CRd |    |    |    | coproc |    |   |   | imm8 |   |   |   |   |   |   |   |

#### Offset variant

Applies when **P == 1 && W == 0**.

STC2{L}{<c>}{<q>} <coproc>, <CRd>, [<Rn>{, #+/-<imm>}]

#### Post-indexed variant

Applies when **P == 0 && W == 1**.

STC2{L}{<c>}{<q>} <coproc>, <CRd>, [<Rn>], #+/-<imm>

#### Pre-indexed variant

Applies when **P == 1 && W == 1**.

STC2{L}{<c>}{<q>} <coproc>, <CRd>, [<Rn>, #+/-<imm>]!

#### Unindexed variant

Applies when **P == 0 && U == 1 && W == 0**.

STC2{L}{<c>}{<q>} <coproc>, <CRd>, [<Rn>], <option>

#### Decode for this encoding

```

1 if P == '0' && U == '0' && D == '1' && W == '0' then SEE "MCRR, MCRR2";
2 if P == '0' && U == '0' && D == '0' && W == '0' then UNDEFINED;
3 if coproc IN '101x' then UNDEFINED;
4 if !HaveMainExt() then UNDEFINED;
5 n = UInt(Rn); cp = UInt(coproc); imm32 = ZeroExtend(imm8:'00', 32);
6 index = (P == '1'); add = (U == '1'); wback = (W == '1');
7 if n == 15 then UNPREDICTABLE;

```

#### Assembler symbols for all encodings

|          |                                                                                                                                                                                                                                                        |
|----------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| L        | If specified, selects the D == 1 form of the encoding. If omitted, selects the D == 0 form.                                                                                                                                                            |
| <c>      | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                                                                                                                                                                               |
| <q>      | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                                                                                                                                                                               |
| <coproc> | Is the name of the coprocessor, encoded in the "coproc" field. The generic coprocessor names are p0 to p15.                                                                                                                                            |
| <CRd>    | Is the coprocessor register to be transferred, encoded in the "CRd" field.                                                                                                                                                                             |
| <Rn>     | Is the general-purpose base register, encoded in the "Rn" field.                                                                                                                                                                                       |
| <option> | Is a coprocessor option, in the range 0 to 255 enclosed in { }, encoded in the "imm8" field.                                                                                                                                                           |
| +/-      | Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in the "U" field. It can have the following values: <ul style="list-style-type: none"> <li>- when U = 0</li> <li>+ when U = 1</li> </ul> |
| <imm>    | Is the immediate offset used for forming the address, a multiple of 4 in the range 0-1020, defaulting to 0 and encoded in the "imm8" field, as <imm>/4.                                                                                                |

#### Operation for all encodings

```

1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 ExecuteCPCheck(cp);
4 if !CoproccAccepted(cp, ThisInstr()) then
5 GenerateCoproccorException();
6 else
7 offset_addr = if add then (R[n] + imm32) else (R[n] - imm32);
8 address = if index then offset_addr else R[n];
9
10 // Determine if the stack pointer limit check should be performed
11 if wback && n == 13 then
12 (limit, applylimit) = LookUpSPLim(LookUpSP());

```

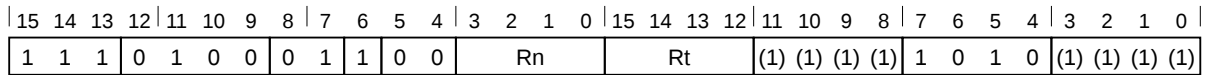
```
13 else
14 applylimit = FALSE;
15
16 // Memory operation only performed if limit not violated
17 if !applylimit || (UInt(offset_addr) >= UInt(limit)) then
18 repeat
19 MemA[address,4] = Coproc_GetWordToStore(cp, ThisInstr());
20 address = address + 4;
21 until Coproc_DoneStoring(cp, ThisInstr());
22
23 // If the stack pointer is being updated a fault will be raised
24 // if the limit is violated
25 if wback then RSPCheck[n] = offset_addr;
```

### C2.4.178 STL

Store-Release Word. Store Release Word stores a word from a register to memory. The instruction also has memory ordering semantics.

#### T1

Armv8-M



#### T1 variant

STL{<c>}{<q>} <Rt>, [<Rn>]

#### Decode for this encoding

```

1 t = UInt(Rt); n = UInt(Rn);
2 if t IN {13,15} || n == 15 then UNPREDICTABLE;

```

#### Assembler symbols for all encodings

- <c> See [C1.2.5 Standard assembler syntax fields](#) on page 309.
- <q> See [C1.2.5 Standard assembler syntax fields](#) on page 309.
- <Rt> Is the general-purpose register to be transferred, encoded in the "Rt" field.
- <Rn> Is the general-purpose base register, encoded in the "Rn" field.

#### Operation for all encodings

```

1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 address = R[n];
4 MemO[address, 4] = R[t];

```



### C2.4.179 STLB

Store-Release Byte. Store Release Byte stores a byte from a register to memory. The instruction also has memory ordering semantics.

#### T1

Armv8-M

|    |    |    |    |    |    |   |   |   |   |   |   |    |   |   |   |    |    |    |    |     |     |     |     |   |   |   |   |     |     |     |     |
|----|----|----|----|----|----|---|---|---|---|---|---|----|---|---|---|----|----|----|----|-----|-----|-----|-----|---|---|---|---|-----|-----|-----|-----|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3  | 2 | 1 | 0 | 15 | 14 | 13 | 12 | 11  | 10  | 9   | 8   | 7 | 6 | 5 | 4 | 3   | 2   | 1   | 0   |
| 1  | 1  | 1  | 0  | 1  | 0  | 0 | 0 | 1 | 1 | 0 | 0 | Rn |   |   |   | Rt |    |    |    | (1) | (1) | (1) | (1) | 1 | 0 | 0 | 0 | (1) | (1) | (1) | (1) |

#### T1 variant

STLB{<c>}{<q>} <Rt>, [<Rn>]

#### Decode for this encoding

```
1 t = UInt(Rt); n = UInt(Rn);
2 if t IN {13,15} || n == 15 then UNPREDICTABLE;
```

#### Assembler symbols for all encodings

<c> See [C1.2.5 Standard assembler syntax fields](#) on page 309.  
 <q> See [C1.2.5 Standard assembler syntax fields](#) on page 309.  
 <Rt> Is the general-purpose register to be transferred, encoded in the "Rt" field.  
 <Rn> Is the general-purpose base register, encoded in the "Rn" field.

#### Operation for all encodings

```
1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 address = R[n];
4 MemO[address, 1] = R[t][7:0];
```

## C2.4.180 STLEX

Store-Release Exclusive Word. Store Release Exclusive Word stores a word from a register to memory if the executing PE has exclusive access to the memory addressed. The instruction also has memory ordering semantics.

### T1

Armv8-M

|    |    |    |    |    |    |   |   |   |   |   |   |    |   |   |   |    |    |    |    |     |     |     |     |   |   |   |   |    |   |   |   |
|----|----|----|----|----|----|---|---|---|---|---|---|----|---|---|---|----|----|----|----|-----|-----|-----|-----|---|---|---|---|----|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3  | 2 | 1 | 0 | 15 | 14 | 13 | 12 | 11  | 10  | 9   | 8   | 7 | 6 | 5 | 4 | 3  | 2 | 1 | 0 |
| 1  | 1  | 1  | 0  | 1  | 0  | 0 | 0 | 1 | 1 | 0 | 0 | Rn |   |   |   | Rt |    |    |    | (1) | (1) | (1) | (1) | 1 | 1 | 1 | 0 | Rd |   |   |   |

### T1 variant

STLEX{<c>}{<q>} <Rd>, <Rt>, [<Rn>]

### Decode for this encoding

```

1 d = UInt(Rd); t = UInt(Rt); n = UInt(Rn);
2 if d IN {13,15} || t IN {13,15} || n == 15 then UNPREDICTABLE;
3 if d == n || d == t then UNPREDICTABLE;

```

### CONSTRAINED UNPREDICTABLE behavior

If  $d == t$ , then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction executes but the value stored is UNKNOWN.

### CONSTRAINED UNPREDICTABLE behavior

If  $d == n$ , then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction performs the store to an UNKNOWN address.

### Assembler symbols for all encodings

|      |                                                                                                                                                                                                                                                                                                        |
|------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <c>  | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                                                                                                                                                                                                                               |
| <q>  | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                                                                                                                                                                                                                               |
| <Rd> | Is the destination general-purpose register into which the status result of the store exclusive is written, encoded in the "Rd" field. The value returned is: <ol style="list-style-type: none"> <li>1 If the operation fails to update memory.</li> <li>0 If the operation updates memory.</li> </ol> |
| <Rt> | Is the general-purpose register to be transferred, encoded in the "Rt" field.                                                                                                                                                                                                                          |
| <Rn> | Is the general-purpose base register, encoded in the "Rn" field.                                                                                                                                                                                                                                       |

### Operation for all encodings

```

1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 address = R[n];
4 if ExclusiveMonitorsPass(address, 4) then
5 MemO[address, 4] = R[t];
6 R[d] = ZeroExtend('0');

```

```
7 else
8 R[d] = ZeroExtend('1');
```

## C2.4.181 STLEXB

Store-Release Exclusive Byte. Store Release Exclusive Byte stores a byte from a register to memory if the executing PE has exclusive access to the memory addressed. The instruction also has memory ordering semantics.

### T1

Armv8-M

|    |    |    |    |    |    |   |   |   |   |   |   |    |   |   |   |    |    |    |    |     |     |     |     |   |   |   |   |    |   |   |   |
|----|----|----|----|----|----|---|---|---|---|---|---|----|---|---|---|----|----|----|----|-----|-----|-----|-----|---|---|---|---|----|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3  | 2 | 1 | 0 | 15 | 14 | 13 | 12 | 11  | 10  | 9   | 8   | 7 | 6 | 5 | 4 | 3  | 2 | 1 | 0 |
| 1  | 1  | 1  | 0  | 1  | 0  | 0 | 0 | 1 | 1 | 0 | 0 | Rn |   |   |   | Rt |    |    |    | (1) | (1) | (1) | (1) | 1 | 1 | 0 | 0 | Rd |   |   |   |

### T1 variant

STLEXB{<c>}{<q>} <Rd>, <Rt>, [<Rn>]

### Decode for this encoding

```

1 d = UInt(Rd); t = UInt(Rt); n = UInt(Rn);
2 if d IN {13,15} || t IN {13,15} || n == 15 then UNPREDICTABLE;
3 if d == n || d == t then UNPREDICTABLE;

```

### CONSTRAINED UNPREDICTABLE behavior

If  $d == t$ , then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction executes but the value stored is UNKNOWN.

### CONSTRAINED UNPREDICTABLE behavior

If  $d == n$ , then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction performs the store to an UNKNOWN address.

### Assembler symbols for all encodings

|      |                                                                                                                                                                                                                                                                                                        |
|------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <c>  | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                                                                                                                                                                                                                               |
| <q>  | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                                                                                                                                                                                                                               |
| <Rd> | Is the destination general-purpose register into which the status result of the store exclusive is written, encoded in the "Rd" field. The value returned is: <ol style="list-style-type: none"> <li>1 If the operation fails to update memory.</li> <li>0 If the operation updates memory.</li> </ol> |
| <Rt> | Is the general-purpose register to be transferred, encoded in the "Rt" field.                                                                                                                                                                                                                          |
| <Rn> | Is the general-purpose base register, encoded in the "Rn" field.                                                                                                                                                                                                                                       |

### Operation for all encodings

```

1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 address = R[n];
4 if ExclusiveMonitorsPass(address,1) then
5 MemO[address, 1] = R[t][7:0];
6 R[d] = ZeroExtend('0');

```

```
7 else
8 R[d] = ZeroExtend('1');
```

## C2.4.182 STLEXH

Store-Release Exclusive Halfword. Store Release Exclusive Halfword stores a halfword from a register to memory if the executing PE has exclusive access to the memory addressed. The instruction also has memory ordering semantics.

### T1

Armv8-M

|    |    |    |    |    |    |   |   |   |   |   |   |    |   |   |   |    |    |    |    |     |     |     |     |   |   |   |   |    |   |   |   |
|----|----|----|----|----|----|---|---|---|---|---|---|----|---|---|---|----|----|----|----|-----|-----|-----|-----|---|---|---|---|----|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3  | 2 | 1 | 0 | 15 | 14 | 13 | 12 | 11  | 10  | 9   | 8   | 7 | 6 | 5 | 4 | 3  | 2 | 1 | 0 |
| 1  | 1  | 1  | 0  | 1  | 0  | 0 | 0 | 1 | 1 | 0 | 0 | Rn |   |   |   | Rt |    |    |    | (1) | (1) | (1) | (1) | 1 | 1 | 0 | 1 | Rd |   |   |   |

### T1 variant

STLEXH{<c>}{<q>} <Rd>, <Rt>, [<Rn>]

### Decode for this encoding

```

1 d = UInt(Rd); t = UInt(Rt); n = UInt(Rn);
2 if d IN {13,15} || t IN {13,15} || n == 15 then UNPREDICTABLE;
3 if d == n || d == t then UNPREDICTABLE;

```

### CONSTRAINED UNPREDICTABLE behavior

If  $d == t$ , then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction executes but the value stored is UNKNOWN.

### CONSTRAINED UNPREDICTABLE behavior

If  $d == n$ , then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction performs the store to an UNKNOWN address.

### Assembler symbols for all encodings

|      |                                                                                                                                                                                                                                                                                                        |
|------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <c>  | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                                                                                                                                                                                                                               |
| <q>  | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                                                                                                                                                                                                                               |
| <Rd> | Is the destination general-purpose register into which the status result of the store exclusive is written, encoded in the "Rd" field. The value returned is: <ol style="list-style-type: none"> <li>1 If the operation fails to update memory.</li> <li>0 If the operation updates memory.</li> </ol> |
| <Rt> | Is the general-purpose register to be transferred, encoded in the "Rt" field.                                                                                                                                                                                                                          |
| <Rn> | Is the general-purpose base register, encoded in the "Rn" field.                                                                                                                                                                                                                                       |

### Operation for all encodings

```

1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 address = R[n];
4 if ExclusiveMonitorsPass(address,2) then
5 MemO[address, 2] = R[t][15:0];

```

```
6 R[d] = ZeroExtend('0');
7 else
8 R[d] = ZeroExtend('1');
```

### C2.4.183 STLH

Store-Release Halfword. Store Release Halfword stores a halfword from a register to memory. The instruction also has memory ordering semantics.

#### T1

Armv8-M

|    |    |    |    |    |    |   |   |   |   |   |   |    |   |   |   |    |    |    |    |     |     |     |     |   |   |   |   |     |     |     |     |
|----|----|----|----|----|----|---|---|---|---|---|---|----|---|---|---|----|----|----|----|-----|-----|-----|-----|---|---|---|---|-----|-----|-----|-----|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3  | 2 | 1 | 0 | 15 | 14 | 13 | 12 | 11  | 10  | 9   | 8   | 7 | 6 | 5 | 4 | 3   | 2   | 1   | 0   |
| 1  | 1  | 1  | 0  | 1  | 0  | 0 | 0 | 1 | 1 | 0 | 0 | Rn |   |   |   | Rt |    |    |    | (1) | (1) | (1) | (1) | 1 | 0 | 0 | 1 | (1) | (1) | (1) | (1) |

#### T1 variant

STLH{<c>}{<q>} <Rt>, [<Rn>]

#### Decode for this encoding

```
1 t = UInt(Rt); n = UInt(Rn);
2 if t IN {13,15} || n == 15 then UNPREDICTABLE;
```

#### Assembler symbols for all encodings

<c> See [C1.2.5 Standard assembler syntax fields](#) on page 309.  
 <q> See [C1.2.5 Standard assembler syntax fields](#) on page 309.  
 <Rt> Is the general-purpose register to be transferred, encoded in the "Rt" field.  
 <Rn> Is the general-purpose base register, encoded in the "Rn" field.

#### Operation for all encodings

```
1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 address = R[n];
4 MemO[address, 2] = R[t][15:0];
```

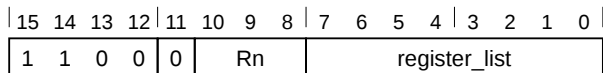


### C2.4.184 STM, STMIA, STMEA

**Store Multiple.** Store Multiple stores multiple registers to consecutive memory locations using an address from a base register. The consecutive memory locations start at this address, and the address just above the last of those locations can optionally be written back to the base register.

#### T1

Armv8-M



#### T1 variant

```
STM{IA}{<c>}{<q>} <Rn>!, <registers>
 // Preferred syntax
STMEA{<c>}{<q>} <Rn>!, <registers>
 // Alternate syntax, Empty Ascending stack
```

#### Decode for this encoding

```
1 n = UInt(Rn); registers = '00000000':register_list; wback = TRUE;
2 if BitCount(registers) < 1 then UNPREDICTABLE;
```

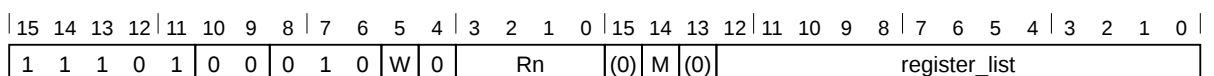
#### CONSTRAINED UNPREDICTABLE behavior

If `BitCount(registers) < 1`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction operates as an STM with the same addressing mode but targeting an unspecified set of registers. These registers might include R15. If the instruction specifies writeback, the modification to the base address on writeback might differ from the number of registers stored.

#### T2

Armv8-M Main Extension only



#### T2 variant

```
STM{IA}{<c>}.W <Rn>{!}, <registers>
 // Preferred syntax
 // if <Rn>, '!' and <registers> can be represented in T1
STMEA{<c>}.W <Rn>{!}, <registers>
 // Alternate syntax
 // Empty Ascending stack
 // if <Rn>, '!' and <registers> can be represented in T1
STM{IA}{<c>}{<q>} <Rn>{!}, <registers>
 // Preferred syntax
STMEA{<c>}{<q>} <Rn>{!}, <registers>
 // Alternate syntax, Empty Ascending stack
```

### Decode for this encoding

```
1 if !HaveMainExt() then UNDEFINED;
2 n = UInt(Rn); registers = '0':M:'0':register_list; wback = (W == '1');
3 if n == 15 || BitCount(registers) < 2 then UNPREDICTABLE;
4 if wback && registers[n] == '1' then UNPREDICTABLE;
```

### CONSTRAINED UNPREDICTABLE behavior

If `BitCount(registers) < 1`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction operates as an STM with the same addressing mode but targeting an unspecified set of registers. These registers might include R15. If the instruction specifies writeback, the modification to the base address on writeback might differ from the number of registers stored.

### CONSTRAINED UNPREDICTABLE behavior

If `BitCount(registers) == 1`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes as described, with no change to its behavior and no additional side effects.
- The instruction operates as an STM with the same addressing mode but targeting an unspecified set of registers. These registers might include R15.

### CONSTRAINED UNPREDICTABLE behavior

If `wback && registers<n> == '1'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction executes but the value stored for the base register is UNKNOWN.

### Assembler symbols for all encodings

|             |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                            |
|-------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| IA          | Is an optional suffix for the Increment After form.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                        |
| <c>         | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                   |
| <q>         | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                   |
| <Rn>        | Is the general-purpose base register, encoded in the "Rn" field.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |
| !           | The address adjusted by the size of the data loaded is written back to the base register. If specified, it is encoded in the "W" field as 1, otherwise this field defaults to 0.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |
| <registers> | For encoding T1: is a list of one or more registers to be stored, separated by commas and surrounded by { and }. The registers in the list must be in the range R0-R7, encoded in the "register_list" field. If the base register is not the lowest-numbered register in the list, such an instruction stores an UNKNOWN value for the base register.<br>For encoding T2: is a list of one or more registers to be stored, separated by commas and surrounded by { and }. The registers in the list must be in the range R0-R12, encoded in the "register_list" field, and can optionally contain the LR. If the LR is in the list, the "M" field is set to 1, otherwise it defaults to 0. |

### Operation for all encodings

```
1 if ConditionPassed() then
```

```
2 EncodingSpecificOperations();
3 address = R[n];
4 endAddress = R[n] + 4*BitCount(registers);
5
6 // Determine if the stack pointer limit should be checked
7 if n == 13 && wback && registers[n] == '0' then
8 (limit, applylimit) = LookUpSPLim(LookUpSP());
9 doOperation = (!applylimit || (UInt(endAddress) >= UInt(limit)));
10 else
11 doOperation = TRUE;
12
13 for i = 0 to 14
14 // Memory operation only performed if limit not violated
15 if registers[i] == '1' && doOperation then
16 if i == n && wback && i != LowestSetBit(registers) then
17 MemA[address,4] = bits(32) UNKNOWN; // encoding T1 only
18 else
19 MemA[address,4] = R[i];
20 address = address + 4;
21
22 // If the stack pointer is being updated a fault will be raised if
23 // the limit is violated
24 if wback then RSPCheck[n] = endAddress;
```

## C2.4.185 STMDB, STMFD

Store Multiple Decrement Before (Full Descending). Store Multiple Decrement Before stores multiple registers to consecutive memory locations using an address from a base register. The consecutive memory locations end just below this address, and the address of the first of those locations can optionally be written back to the base register.

This instruction is used by the alias [PUSH \(multiple registers\)](#).

### T1

Armv8-M Main Extension only

|    |    |    |    |    |    |   |   |   |   |   |   |    |   |   |   |     |    |     |               |    |    |   |   |   |   |   |   |   |   |   |   |
|----|----|----|----|----|----|---|---|---|---|---|---|----|---|---|---|-----|----|-----|---------------|----|----|---|---|---|---|---|---|---|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3  | 2 | 1 | 0 | 15  | 14 | 13  | 12            | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 1  | 1  | 1  | 0  | 1  | 0  | 0 | 1 | 0 | 0 | W | 0 | Rn |   |   |   | (0) | M  | (0) | register_list |    |    |   |   |   |   |   |   |   |   |   |   |

### T1 variant

```
STMDB<{c}>{<q>} <Rn>{!}, <registers>
// Preferred syntax
STMFD<{c}>{<q>} <Rn>{!}, <registers>
// Alternate syntax, Full Descending stack
```

### Decode for this encoding

```
1 if !HaveMainExt() then UNDEFINED;
2 n = UInt(Rn); registers = '0':M:'0':register_list; wback = (W == '1');
3 if n == 15 || BitCount(registers) < 2 then UNPREDICTABLE;
4 if wback && registers[n] == '1' then UNPREDICTABLE;
```

### CONSTRAINED UNPREDICTABLE behavior

If `BitCount(registers) < 1`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction operates as an STM with the same addressing mode but targeting an unspecified set of registers. These registers might include R15. If the instruction specifies writeback, the modification to the base address on writeback might differ from the number of registers stored.

### CONSTRAINED UNPREDICTABLE behavior

If `wback && registers<n> == '1'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction executes but the value stored for the base register is UNKNOWN.

### CONSTRAINED UNPREDICTABLE behavior

If `BitCount(registers) == 1`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes as described, with no change to its behavior and no additional side effects.
- The instruction operates as an STM with the same addressing mode but targeting an unspecified set of registers. These registers might include R15.

## T2

Armv8-M

|    |    |    |    |    |    |   |   |               |   |   |   |   |   |   |   |
|----|----|----|----|----|----|---|---|---------------|---|---|---|---|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7             | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 1  | 0  | 1  | 1  | 0  | 1  | 0 | M | register_list |   |   |   |   |   |   |   |

### T2 variant

STMDB{<c>}{<q>} SP!, <registers>

### Decode for this encoding

```
1 n = 13; wback = TRUE;
2 registers = '0':M:'000000':register_list;
3 if BitCount(registers) < 1 then UNPREDICTABLE;
```

### CONSTRAINED UNPREDICTABLE behavior

If `BitCount(registers) < 1`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction operates as an STM with the same addressing mode but targeting an unspecified set of registers. These registers might include R15. If the instruction specifies writeback, the modification to the base address on writeback might differ from the number of registers stored.

### Assembler symbols for all encodings

|             |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                               |
|-------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <c>         | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                      |
| <q>         | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                      |
| <Rn>        | Is the general-purpose base register, encoded in the "Rn" field.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                              |
| !           | The address adjusted by the size of the data loaded is written back to the base register. If specified, it is encoded in the "W" field as 1, otherwise this field defaults to 0.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                              |
| <registers> | For encoding T1: is a list of one or more registers to be stored, separated by commas and surrounded by { and }. The registers in the list must be in the range R0-R12, encoded in the "register_list" field, and can optionally contain the LR. If the LR is in the list, the "M" field is set to 1, otherwise it defaults to 0.<br>For encoding T2: is a list of one or more registers to be stored, separated by commas and surrounded by { and }. The registers in the list must be in the range R0-R7, encoded in the "register_list" field, and can optionally include the LR. If the LR is in the list, the "M" field is set to 1, otherwise this field defaults to 0. |

### Operation for all encodings

```
1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 address = R[n] - 4*BitCount(registers);
4 if n == 13 && wback then
5 (limit, applylimit) = LookUpSPLim(LookUpSP());
6 else
7 applylimit = FALSE;
8
9 for i = 0 to 14
10 // If R[n] is the SP, memory operation only performed if limit not violated
11 if registers[i] == '1' && (!applylimit || (UInt(address) >= UInt(limit))) then
12 MemA[address,4] = R[i];
13 address = address + 4;
14
15 // If R[n] is the SP, stack pointer update will raise a fault if limit violated
```

```
16 if wback then RSPCheck[n] = R[n] - 4*BitCount(registers);
```

### C2.4.186 STR (immediate)

Store Register (immediate). Store Register (immediate) calculates an address from a base register value and an immediate offset, and stores a word from a register to memory. It can use offset, post-indexed, or pre-indexed addressing.

This instruction is used by the alias **PUSH (single register)**.

#### T1

Armv8-M

|    |    |    |    |    |      |   |   |   |    |   |   |    |   |   |   |
|----|----|----|----|----|------|---|---|---|----|---|---|----|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10   | 9 | 8 | 7 | 6  | 5 | 4 | 3  | 2 | 1 | 0 |
| 0  | 1  | 1  | 0  | 0  | imm5 |   |   |   | Rn |   |   | Rt |   |   |   |

#### T1 variant

STR{<c>}{<q>} <Rt>, [<Rn> {, #<+><imm>}]

#### Decode for this encoding

```
1 t = UInt(Rt); n = UInt(Rn); imm32 = ZeroExtend(imm5:'00', 32);
2 index = TRUE; add = TRUE; wback = FALSE;
```

#### T2

Armv8-M

|    |    |    |    |    |    |   |   |      |   |   |   |   |   |   |   |
|----|----|----|----|----|----|---|---|------|---|---|---|---|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7    | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 1  | 0  | 0  | 1  | 0  | Rt |   |   | imm8 |   |   |   |   |   |   |   |

#### T2 variant

STR{<c>}{<q>} <Rt>, [SP{, #<+><imm>}]

#### Decode for this encoding

```
1 t = UInt(Rt); n = 13; imm32 = ZeroExtend(imm8:'00', 32);
2 index = TRUE; add = TRUE; wback = FALSE;
```

#### T3

Armv8-M Main Extension only

|    |    |    |    |    |    |   |   |   |   |   |   |            |   |   |    |    |       |    |    |    |    |   |   |   |   |   |   |   |   |   |   |
|----|----|----|----|----|----|---|---|---|---|---|---|------------|---|---|----|----|-------|----|----|----|----|---|---|---|---|---|---|---|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3          | 2 | 1 | 0  | 15 | 14    | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 1  | 1  | 1  | 1  | 1  | 0  | 0 | 0 | 1 | 1 | 0 | 0 | Rn != 1111 |   |   | Rt |    | imm12 |    |    |    |    |   |   |   |   |   |   |   |   |   |   |

#### T3 variant

STR{<c>}.W <Rt>, [<Rn> {, #<+><imm>}]  
 // <Rt>, <Rn>, <imm> can be represented in T1 or T2  
 STR{<c>}{<q>} <Rt>, [<Rn> {, #<+><imm>}]

#### Decode for this encoding

```
1 if Rn == '1111' then UNDEFINED;
2 if !HaveMainExt() then UNDEFINED;
3 t = UInt(Rt); n = UInt(Rn); imm32 = ZeroExtend(imm12, 32);
4 index = TRUE; add = TRUE; wback = FALSE;
5 if t == 15 then UNPREDICTABLE;
```





<imm> For the post-indexed or pre-indexed variant: is an 8-bit unsigned immediate byte offset, in the range 0 to 255, encoded in the "imm8" field.  
For encoding T1: is the optional positive unsigned immediate byte offset, a multiple of 4 in the range 0 to 124, defaulting to 0 and encoded in the "imm5" field as <imm>/4.  
For encoding T2: is the optional positive unsigned immediate byte offset, a multiple of 4, in the range 0 to 1020, defaulting to 0 and encoded in the "imm8" field as <imm>/4.  
For encoding T3: is an optional 12-bit unsigned immediate byte offset, in the range 0 to 4095, defaulting to 0 and encoded in the "imm12" field.  
For encoding T4: is an optional 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 and encoded in the "imm8" field.

### Operation for all encodings

```
1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 offset_addr = if add then (R[n] + imm32) else (R[n] - imm32);
4 address = if index then offset_addr else R[n];
5
6 // Determine if the stack pointer limit should be checked
7 if n == 13 && wback then
8 (limit, applylimit) = LookUpSPLim(LookUpSP());
9 else
10 applylimit = FALSE;
11 // Memory operation only performed if limit not violated
12 if !applylimit || (UInt(offset_addr) >= UInt(limit)) then
13 MemU[address,4] = R[t];
14
15 // If the stack pointer is being updated a fault will be raised if
16 // the limit is violated
17 if wback then RSPCheck[n] = offset_addr;
```

### C2.4.187 STR (register)

Store Register (register). Store Register (register) calculates an address from a base register value and an offset register value, stores a word from a register to memory. The offset register value can be shifted left by 0, 1, 2, or 3 bits.

#### T1

Armv8-M

|    |    |    |    |    |    |   |   |    |   |    |   |   |   |   |    |
|----|----|----|----|----|----|---|---|----|---|----|---|---|---|---|----|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7  | 6 | 5  | 4 | 3 | 2 | 1 | 0  |
| 0  | 1  | 0  | 1  | 0  | 0  | 0 |   | Rm |   | Rn |   |   |   |   | Rt |

#### T1 variant

STR{<c>}{<q>} <Rt>, [<Rn>, {+}<Rm>]

#### Decode for this encoding

```
1 t = UInt(Rt); n = UInt(Rn); m = UInt(Rm);
2 index = TRUE; add = TRUE; wback = FALSE;
3 (shift_t, shift_n) = (SRTYPE_LSL, 0);
```

#### T2

Armv8-M Main Extension only

|    |    |    |    |    |    |   |   |   |   |   |   |            |   |    |   |    |    |    |    |    |    |      |   |    |   |   |   |   |   |   |   |
|----|----|----|----|----|----|---|---|---|---|---|---|------------|---|----|---|----|----|----|----|----|----|------|---|----|---|---|---|---|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3          | 2 | 1  | 0 | 15 | 14 | 13 | 12 | 11 | 10 | 9    | 8 | 7  | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 1  | 1  | 1  | 1  | 1  | 0  | 0 | 0 | 0 | 1 | 0 | 0 | Rn != 1111 |   | Rt |   | 0  | 0  | 0  | 0  | 0  | 0  | imm2 |   | Rm |   |   |   |   |   |   |   |

#### T2 variant

STR{<c>}.W <Rt>, [<Rn>, {+}<Rm>]  
 // <Rt>, <Rn>, <Rm> can be represented in T1  
 STR{<c>}{<q>} <Rt>, [<Rn>, {+}<Rm>{, LSL #<imm>}]

#### Decode for this encoding

```
1 if Rn == '1111' then UNDEFINED;
2 if !HaveMainExt() then UNDEFINED;
3 t = UInt(Rt); n = UInt(Rn); m = UInt(Rm);
4 index = TRUE; add = TRUE; wback = FALSE;
5 (shift_t, shift_n) = (SRTYPE_LSL, UInt(imm2));
6 if t == 15 || m IN {13,15} then UNPREDICTABLE;
```

#### Assembler symbols for all encodings

|       |                                                                                                                                                                                   |
|-------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <c>   | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                                                                                                          |
| <q>   | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                                                                                                          |
| <Rt>  | Is the general-purpose register to be transferred, encoded in the "Rt" field.                                                                                                     |
| <Rn>  | Is the general-purpose base register, encoded in the "Rn" field.                                                                                                                  |
| +     | Specifies the index register is added to the base register.                                                                                                                       |
| <Rm>  | Is the general-purpose index register, encoded in the "Rm" field.                                                                                                                 |
| <imm> | If present, the size of the left shift to apply to the value from <Rm>, in the range 1-3. <imm> is encoded in imm2. If absent, no shift is specified and imm2 is encoded as 0b00. |

#### Operation for all encodings

```
1 if ConditionPassed() then
```

## Chapter C2. Instruction Specification

### C2.4. Alphabetical list of instructions

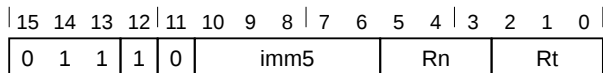
```
2 EncodingSpecificOperations();
3 offset = Shift(R[m], shift_t, shift_n, APSR.C);
4 address = R[n] + offset;
5 MemU[address,4] = R[t];
```

### C2.4.188 STRB (immediate)

Store Register Byte (immediate). Store Register Byte (immediate) calculates an address from a base register value and an immediate offset, and stores a byte from a register to memory. It can use offset, post-indexed, or pre-indexed addressing.

#### T1

Armv8-M



#### T1 variant

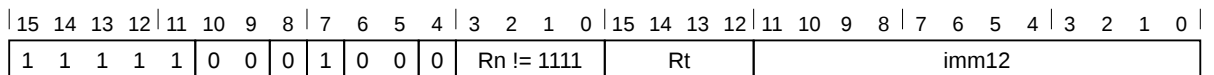
STRB{<c>}{<q>} <Rt>, [<Rn> {, #(<+>)<imm>}]

#### Decode for this encoding

```
1 t = UInt(Rt); n = UInt(Rn); imm32 = ZeroExtend(imm5, 32);
2 index = TRUE; add = TRUE; wback = FALSE;
```

#### T2

Armv8-M Main Extension only



#### T2 variant

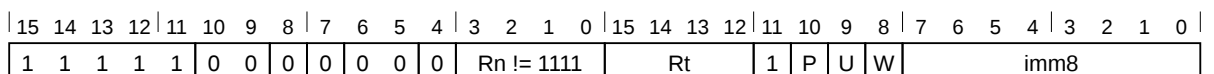
STRB{<c>}.W <Rt>, [<Rn> {, #(<+>)<imm>}]  
 // <Rt>, <Rn>, <imm> can be represented in T1  
 STRB{<c>}{<q>} <Rt>, [<Rn> {, #(<+>)<imm>}]

#### Decode for this encoding

```
1 if Rn == '1111' then UNDEFINED;
2 if !HaveMainExt() then UNDEFINED;
3 t = UInt(Rt); n = UInt(Rn); imm32 = ZeroExtend(imm12, 32);
4 index = TRUE; add = TRUE; wback = FALSE;
5 if t IN {13,15} then UNPREDICTABLE;
```

#### T3

Armv8-M Main Extension only



#### Offset variant

Applies when **P == 1 && U == 0 && W == 0.**

STRB{<c>}{<q>} <Rt>, [<Rn> {, #-<imm>}]

#### Post-indexed variant

Applies when **P == 0 && W == 1.**

STRB{<c>}{<q>} <Rt>, [<Rn>], #{+/-}<imm>

### Pre-indexed variant

Applies when **P == 1 && W == 1**.

STRB{<c>}{<q>} <Rt>, [<Rn>, #{+/-}<imm>]!

### Decode for this encoding

```

1 if P == '1' && U == '1' && W == '0' then SEE STRBT;
2 if Rn == '1111' || (P == '0' && W == '0') then UNDEFINED;
3 if !HaveMainExt() then UNDEFINED;
4 t = UInt(Rt); n = UInt(Rn); imm32 = ZeroExtend(imm8, 32);
5 index = (P == '1'); add = (U == '1'); wback = (W == '1');
6 if t IN {13,15} || (wback && n == t) then UNPREDICTABLE;

```

### CONSTRAINED UNPREDICTABLE behavior

If `wback && n == t`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction executes but the value stored is UNKNOWN.

### Assembler symbols for all encodings

|       |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                   |
|-------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <c>   | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                          |
| <q>   | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                          |
| <Rt>  | Is the general-purpose register to be transferred, encoded in the "Rt" field.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                     |
| <Rn>  | Is the general-purpose base register, encoded in the "Rn" field.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                  |
| +/-   | Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in the "U" field. It can have the following values: <ul style="list-style-type: none"> <li>- when U = 0</li> <li>+ when U = 1</li> </ul>                                                                                                                                                                                                                                                                                                                                            |
| +     | Specifies the offset is added to the base register.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                               |
| <imm> | For the post-indexed or pre-indexed variant: is an 8-bit unsigned immediate byte offset, in the range 0 to 255, encoded in the "imm8" field.<br>For encoding T1: is an optional 5-bit unsigned immediate byte offset, in the range 0 to 31, defaulting to 0 and encoded in the "imm5" field.<br>For encoding T2: is an optional 12-bit unsigned immediate byte offset, in the range 0 to 4095, defaulting to 0 and encoded in the "imm12" field.<br>For encoding T3: is an optional 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 and encoded in the "imm8" field. |

### Operation for all encodings

```

1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 offset_addr = if add then (R[n] + imm32) else (R[n] - imm32);
4 address = if index then offset_addr else R[n];
5
6 // Determine if the stack pointer limit should be checked
7 if n == 13 && wback then
8 (limit, applylimit) = LookUpSPLim(LookUpSP());
9 else
10 applylimit = FALSE;
11 // Memory operation only performed if limit not violated
12 if !applylimit || (UInt(offset_addr) >= UInt(limit)) then

```

```
13 MemU[address,1] = R[t][7:0];
14
15 // If the stack pointer is being updated a fault will be raised if
16 // the limit is violated
17 if wback then RSPCheck[n] = offset_addr;
```

### C2.4.189 STRB (register)

Store Register Byte (register). Store Register Byte (register) calculates an address from a base register value and an offset register value, and stores a byte from a register to memory. The offset register value can be shifted left by 0, 1, 2, or 3 bits.

#### T1

Armv8-M

|    |    |    |    |    |    |   |   |    |   |    |   |   |   |   |    |
|----|----|----|----|----|----|---|---|----|---|----|---|---|---|---|----|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7  | 6 | 5  | 4 | 3 | 2 | 1 | 0  |
| 0  | 1  | 0  | 1  | 0  | 1  | 0 |   | Rm |   | Rn |   |   |   |   | Rt |

#### T1 variant

STRB{<c>}{<q>} <Rt>, [<Rn>, {+}<Rm>]

#### Decode for this encoding

```
1 t = UInt(Rt); n = UInt(Rn); m = UInt(Rm);
2 index = TRUE; add = TRUE; wback = FALSE;
3 (shift_t, shift_n) = (SRTYPE_LSL, 0);
```

#### T2

Armv8-M Main Extension only

|    |    |    |    |    |    |   |   |   |   |   |   |            |   |    |   |    |    |    |    |    |    |      |   |    |   |   |   |   |   |   |   |
|----|----|----|----|----|----|---|---|---|---|---|---|------------|---|----|---|----|----|----|----|----|----|------|---|----|---|---|---|---|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3          | 2 | 1  | 0 | 15 | 14 | 13 | 12 | 11 | 10 | 9    | 8 | 7  | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 1  | 1  | 1  | 1  | 1  | 0  | 0 | 0 | 0 | 0 | 0 | 0 | Rn != 1111 |   | Rt |   | 0  | 0  | 0  | 0  | 0  | 0  | imm2 |   | Rm |   |   |   |   |   |   |   |

#### T2 variant

STRB{<c>}.W <Rt>, [<Rn>, {+}<Rm>]  
 // <Rt>, <Rn>, <Rm> can be represented in T1  
 STRB{<c>}{<q>} <Rt>, [<Rn>, {+}<Rm>{, LSL #<imm>}]

#### Decode for this encoding

```
1 if Rn == '1111' then UNDEFINED;
2 if !HaveMainExt() then UNDEFINED;
3 t = UInt(Rt); n = UInt(Rn); m = UInt(Rm);
4 index = TRUE; add = TRUE; wback = FALSE;
5 (shift_t, shift_n) = (SRTYPE_LSL, UInt(imm2));
6 if t IN {13,15} || m IN {13,15} then UNPREDICTABLE;
```

#### Assembler symbols for all encodings

|       |                                                                                                                                                                                   |
|-------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <c>   | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                                                                                                          |
| <q>   | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                                                                                                          |
| <Rt>  | Is the general-purpose register to be transferred, encoded in the "Rt" field.                                                                                                     |
| <Rn>  | Is the general-purpose base register, encoded in the "Rn" field.                                                                                                                  |
| +     | Specifies the index register is added to the base register.                                                                                                                       |
| <Rm>  | Is the general-purpose index register, encoded in the "Rm" field.                                                                                                                 |
| <imm> | If present, the size of the left shift to apply to the value from <Rm>, in the range 1-3. <imm> is encoded in imm2. If absent, no shift is specified and imm2 is encoded as 0b00. |

#### Operation for all encodings

```
1 if ConditionPassed() then
```

## Chapter C2. Instruction Specification

### C2.4. Alphabetical list of instructions

```
2 EncodingSpecificOperations();
3 offset = Shift(R[m], shift_t, shift_n, APSR.C);
4 address = R[n] + offset;
5 MemU[address,1] = R[t][7:0];
```



## C2.4.190 STRBT

Store Register Byte Unprivileged. Store Register Byte Unprivileged calculates an address from a base register value and an immediate offset, and stores a byte from a register to memory. When privileged software uses an [STRBT](#) instruction, the memory access is restricted as if the software was unprivileged.

### T1

Armv8-M Main Extension only

|    |    |    |    |    |    |   |   |   |   |   |   |            |   |    |   |    |    |    |    |      |    |   |   |   |   |   |   |   |   |   |   |
|----|----|----|----|----|----|---|---|---|---|---|---|------------|---|----|---|----|----|----|----|------|----|---|---|---|---|---|---|---|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3          | 2 | 1  | 0 | 15 | 14 | 13 | 12 | 11   | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 1  | 1  | 1  | 1  | 1  | 0  | 0 | 0 | 0 | 0 | 0 | 0 | Rn != 1111 |   | Rt |   | 1  | 1  | 1  | 0  | imm8 |    |   |   |   |   |   |   |   |   |   |   |

### T1 variant

STRBT{<c>}{<q>} <Rt>, [<Rn> {, #<+><imm>}]

### Decode for this encoding

```

1 if Rn == '1111' then UNDEFINED;
2 if !HaveMainExt() then UNDEFINED;
3 t = UInt(Rt); n = UInt(Rn); postindex = FALSE; add = TRUE;
4 register_form = FALSE; imm32 = ZeroExtend(imm8, 32);
5 if t IN {13,15} then UNPREDICTABLE;

```

### Assembler symbols for all encodings

|       |                                                                                                                              |
|-------|------------------------------------------------------------------------------------------------------------------------------|
| <c>   | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                                                     |
| <q>   | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                                                     |
| <Rt>  | Is the general-purpose register to be transferred, encoded in the "Rt" field.                                                |
| <Rn>  | Is the general-purpose base register, encoded in the "Rn" field.                                                             |
| +     | Specifies the offset is added to the base register.                                                                          |
| <imm> | Is an optional 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 and encoded in the "imm8" field. |

### Operation for all encodings

```

1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 address = R[n] + imm32;
4 MemU_unpriv[address,1] = R[t][7:0];

```

### C2.4.191 STRD (immediate)

Store Register Dual (immediate). Store Register Dual (immediate) calculates an address from a base register value and an immediate offset, and stores two words from two registers to memory. It can use offset, post-indexed, or pre-indexed addressing.

#### T1

Armv8-M Main Extension only

|    |    |    |    |    |    |   |   |   |   |   |   |            |   |   |   |    |    |    |    |     |    |   |   |      |   |   |   |   |   |   |   |
|----|----|----|----|----|----|---|---|---|---|---|---|------------|---|---|---|----|----|----|----|-----|----|---|---|------|---|---|---|---|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3          | 2 | 1 | 0 | 15 | 14 | 13 | 12 | 11  | 10 | 9 | 8 | 7    | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 1  | 1  | 1  | 0  | 1  | 0  | 0 | P | U | 1 | W | 0 | Rn != 1111 |   |   |   | Rt |    |    |    | Rt2 |    |   |   | imm8 |   |   |   |   |   |   |   |

#### Offset variant

Applies when **P == 1 && W == 0**.

STRD{<c>}{<q>} <Rt>, <Rt2>, [<Rn> {, # {+/-} <imm>}]

#### Post-indexed variant

Applies when **P == 0 && W == 1**.

STRD{<c>}{<q>} <Rt>, <Rt2>, [<Rn>], # {+/-} <imm>

#### Pre-indexed variant

Applies when **P == 1 && W == 1**.

STRD{<c>}{<q>} <Rt>, <Rt2>, [<Rn>, # {+/-} <imm>]!

#### Decode for this encoding

```

1 if P == '0' && W == '0' then SEE "Related encodings";
2 if !HaveMainExt() then UNDEFINED;
3 t = UInt(Rt); t2 = UInt(Rt2); n = UInt(Rn); imm32 = ZeroExtend(imm8:'00', 32);
4 index = (P == '1'); add = (U == '1'); wback = (W == '1');
5 if wback && (n == t || n == t2) then UNPREDICTABLE;
6 if n == 15 || t IN {13,15} || t2 IN {13,15} then UNPREDICTABLE;
```

#### CONSTRAINED UNPREDICTABLE behavior

If `wback && (n == t || n == t2)`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction executes but the value stored is UNKNOWN.

#### Assembler symbols for all encodings

|       |                                                                                                                                                                                                                                                        |
|-------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <c>   | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                                                                                                                                                                               |
| <q>   | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                                                                                                                                                                               |
| <Rt>  | Is the first general-purpose register to be transferred, encoded in the "Rt" field.                                                                                                                                                                    |
| <Rt2> | Is the second general-purpose register to be transferred, encoded in the "Rt2" field.                                                                                                                                                                  |
| <Rn>  | Is the general-purpose base register, encoded in the "Rn" field.                                                                                                                                                                                       |
| +/-   | Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in the "U" field. It can have the following values: <ul style="list-style-type: none"> <li>- when U = 0</li> <li>+ when U = 1</li> </ul> |

<imm> For the offset variant: is the optional unsigned immediate byte offset, a multiple of 4, in the range 0 to 1020, defaulting to 0 and encoded in the "imm8" field as <imm>/4.  
For the post-indexed and pre-indexed variant: is the unsigned immediate byte offset, a multiple of 4, in the range 0 to 1020, encoded in the "imm8" field as <imm>/4.

### Operation for all encodings

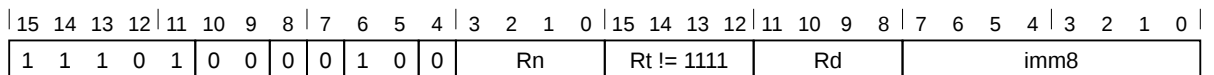
```
1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 offset_addr = if add then (R[n] + imm32) else (R[n] - imm32);
4 address = if index then offset_addr else R[n];
5
6 // Determine if the stack pointer limit should be checked
7 if n == 13 && wback then
8 (limit, applylimit) = LookUpSPLim(LookUpSP());
9 else
10 applylimit = FALSE;
11 // Memory operation only performed if limit not violated
12 if !applylimit || (UInt(offset_addr) >= UInt(limit)) then
13 MemA[address,4] = R[t];
14 MemA[address+4,4] = R[t2];
15
16 // If the stack pointer is being updated a fault will be raised if
17 // the limit is violated
18 if wback then RSPCheck[n] = offset_addr;
```

## C2.4.192 STREX

Store Register Exclusive. Store Register Exclusive calculates an address from a base register value and an immediate offset, and stores a word from a register to memory if the executing PE has exclusive access to the memory addressed.

### T1

Armv8-M



### T1 variant

STREX{<c>}{<q>} <Rd>, <Rt>, [<Rn> {, #<imm>}]

### Decode for this encoding

```

1 d = UInt(Rd); t = UInt(Rt); n = UInt(Rn); imm32 = ZeroExtend(imm8:'00', 32);
2 if t == 15 then SEE "TT";
3 if d IN {13,15} || t == 13 || n == 15 then UNPREDICTABLE;
4 if d == n || d == t then UNPREDICTABLE;
```

### CONSTRAINED UNPREDICTABLE behavior

If  $d == t$ , then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction executes but the value stored is UNKNOWN.

### CONSTRAINED UNPREDICTABLE behavior

If  $d == n$ , then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction performs the store to an UNKNOWN address.

### Assembler symbols for all encodings

|                          |                                                                                                                                                                                                                                                                                                        |
|--------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>&lt;c&gt;</code>   | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                                                                                                                                                                                                                               |
| <code>&lt;q&gt;</code>   | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                                                                                                                                                                                                                               |
| <code>&lt;Rd&gt;</code>  | Is the destination general-purpose register into which the status result of the store exclusive is written, encoded in the "Rd" field. The value returned is: <ul style="list-style-type: none"> <li>1 If the operation fails to update memory.</li> <li>0 If the operation updates memory.</li> </ul> |
| <code>&lt;Rt&gt;</code>  | Is the general-purpose register to be transferred, encoded in the "Rt" field.                                                                                                                                                                                                                          |
| <code>&lt;Rn&gt;</code>  | Is the general-purpose base register, encoded in the "Rn" field.                                                                                                                                                                                                                                       |
| <code>&lt;imm&gt;</code> | The immediate offset added to the value of <code>&lt;Rn&gt;</code> to calculate the address. <code>&lt;imm&gt;</code> can be omitted, meaning an offset of 0. Values are multiples of 4 in the range 0-1020.                                                                                           |

### Operation for all encodings

```

1 if ConditionPassed() then
2 EncodingSpecificOperations();
```

```
3 address = R[n] + imm32;
4 if ExclusiveMonitorsPass(address,4) then
5 MemA[address,4] = R[t];
6 R[d] = ZeroExtend('0');
7 else
8 R[d] = ZeroExtend('1');
```

### C2.4.193 STREXB

Store Register Exclusive Byte. Store Register Exclusive Byte derives an address from a base register value, and stores a byte from a register to memory if the executing PE has exclusive access to the memory addressed.

#### T1

Armv8-M

|    |    |    |    |    |    |   |   |   |   |   |   |   |    |   |    |     |     |     |     |    |    |   |   |   |   |   |   |    |   |   |   |
|----|----|----|----|----|----|---|---|---|---|---|---|---|----|---|----|-----|-----|-----|-----|----|----|---|---|---|---|---|---|----|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2  | 1 | 0  | 15  | 14  | 13  | 12  | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3  | 2 | 1 | 0 |
| 1  | 1  | 1  | 0  | 1  | 0  | 0 | 0 | 1 | 1 | 0 | 0 |   | Rn |   | Rt | (1) | (1) | (1) | (1) | 0  | 1  | 0 | 0 |   |   |   |   | Rd |   |   |   |

#### T1 variant

STREXB{<c>}{<q>} <Rd>, <Rt>, [<Rn>]

#### Decode for this encoding

```

1 d = UInt(Rd); t = UInt(Rt); n = UInt(Rn);
2 if d IN {13,15} || t IN {13,15} || n == 15 then UNPREDICTABLE;
3 if d == n || d == t then UNPREDICTABLE;

```

#### CONSTRAINED UNPREDICTABLE behavior

If  $d == t$ , then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction executes but the value stored is UNKNOWN.

#### CONSTRAINED UNPREDICTABLE behavior

If  $d == n$ , then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction performs the store to an UNKNOWN address.

#### Assembler symbols for all encodings

|      |                                                                                                                                                                                                                                                                                                        |
|------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <c>  | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                                                                                                                                                                                                                               |
| <q>  | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                                                                                                                                                                                                                               |
| <Rd> | Is the destination general-purpose register into which the status result of the store exclusive is written, encoded in the "Rd" field. The value returned is: <ol style="list-style-type: none"> <li>1 If the operation fails to update memory.</li> <li>0 If the operation updates memory.</li> </ol> |
| <Rt> | Is the general-purpose register to be transferred, encoded in the "Rt" field.                                                                                                                                                                                                                          |
| <Rn> | Is the general-purpose base register, encoded in the "Rn" field.                                                                                                                                                                                                                                       |

#### Operation for all encodings

```

1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 address = R[n];
4 if ExclusiveMonitorsPass(address,1) then
5 MemA[address,1] = R[t][7:0];
6 R[d] = ZeroExtend('0');

```

```
7 else
8 R[d] = ZeroExtend('1');
```

## C2.4.194 STREXH

Store Register Exclusive Halfword. Store Register Exclusive Halfword derives an address from a base register value, and stores a halfword from a register to memory if the executing PE has exclusive access to the memory addressed.

### T1

Armv8-M

|    |    |    |    |    |    |   |   |   |   |   |   |    |   |   |   |    |    |    |    |     |     |     |     |   |   |   |   |    |   |   |   |
|----|----|----|----|----|----|---|---|---|---|---|---|----|---|---|---|----|----|----|----|-----|-----|-----|-----|---|---|---|---|----|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3  | 2 | 1 | 0 | 15 | 14 | 13 | 12 | 11  | 10  | 9   | 8   | 7 | 6 | 5 | 4 | 3  | 2 | 1 | 0 |
| 1  | 1  | 1  | 0  | 1  | 0  | 0 | 0 | 1 | 1 | 0 | 0 | Rn |   |   |   | Rt |    |    |    | (1) | (1) | (1) | (1) | 0 | 1 | 0 | 1 | Rd |   |   |   |

### T1 variant

STREXH{<c>}{<q>} <Rd>, <Rt>, [<Rn>]

### Decode for this encoding

```

1 d = UInt(Rd); t = UInt(Rt); n = UInt(Rn);
2 if d IN {13,15} || t IN {13,15} || n == 15 then UNPREDICTABLE;
3 if d == n || d == t then UNPREDICTABLE;

```

### CONSTRAINED UNPREDICTABLE behavior

If  $d == t$ , then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction executes but the value stored is UNKNOWN.

### CONSTRAINED UNPREDICTABLE behavior

If  $d == n$ , then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction performs the store to an UNKNOWN address.

### Assembler symbols for all encodings

|      |                                                                                                                                                                                                                                                                                                        |
|------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <c>  | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                                                                                                                                                                                                                               |
| <q>  | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                                                                                                                                                                                                                               |
| <Rd> | Is the destination general-purpose register into which the status result of the store exclusive is written, encoded in the "Rd" field. The value returned is: <ol style="list-style-type: none"> <li>1 If the operation fails to update memory.</li> <li>0 If the operation updates memory.</li> </ol> |
| <Rt> | Is the general-purpose register to be transferred, encoded in the "Rt" field.                                                                                                                                                                                                                          |
| <Rn> | Is the general-purpose base register, encoded in the "Rn" field.                                                                                                                                                                                                                                       |

### Operation for all encodings

```

1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 address = R[n];
4 if ExclusiveMonitorsPass(address,2) then
5 MemA[address,2] = R[t][15:0];

```



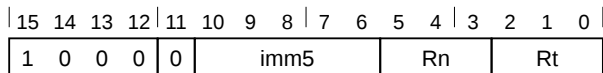
```
6 R[d] = ZeroExtend('0');
7 else
8 R[d] = ZeroExtend('1');
```

### C2.4.195 STRH (immediate)

Store Register Halfword (immediate). Store Register Halfword (immediate) calculates an address from a base register value and an immediate offset, and stores a halfword from a register to memory. It can use offset, post-indexed, or pre-indexed addressing.

#### T1

Armv8-M



#### T1 variant

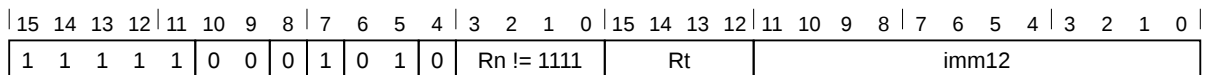
STRH{<c>}{<q>} <Rt>, [<Rn> {, #(+)<imm>}]

#### Decode for this encoding

```
1 t = UInt(Rt); n = UInt(Rn); imm32 = ZeroExtend(imm5:'0', 32);
2 index = TRUE; add = TRUE; wback = FALSE;
```

#### T2

Armv8-M Main Extension only



#### T2 variant

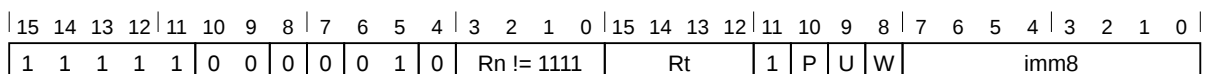
STRH{<c>}.W <Rt>, [<Rn> {, #(+)<imm>}]  
 // <Rt>, <Rn>, <imm> can be represented in T1  
 STRH{<c>}{<q>} <Rt>, [<Rn> {, #(+)<imm>}]

#### Decode for this encoding

```
1 if Rn == '1111' then UNDEFINED;
2 if !HaveMainExt() then UNDEFINED;
3 t = UInt(Rt); n = UInt(Rn); imm32 = ZeroExtend(imm12, 32);
4 index = TRUE; add = TRUE; wback = FALSE;
5 if t IN {13,15} then UNPREDICTABLE;
```

#### T3

Armv8-M Main Extension only



#### Offset variant

Applies when **P == 1** && **U == 0** && **W == 0**.

STRH{<c>}{<q>} <Rt>, [<Rn> {, #-<imm>}]

#### Post-indexed variant

Applies when **P == 0** && **W == 1**.

STRH{<c>}{<q>} <Rt>, [<Rn>], #{+/-}<imm>

### Pre-indexed variant

Applies when **P == 1 && W == 1**.

STRH{<c>}{<q>} <Rt>, [<Rn>, #{+/-}<imm>]!

### Decode for this encoding

```

1 if P == '1' && U == '1' && W == '0' then SEE STRHT;
2 if Rn == '1111' || (P == '0' && W == '0') then UNDEFINED;
3 if !HaveMainExt() then UNDEFINED;
4 t = UInt(Rt); n = UInt(Rn); imm32 = ZeroExtend(imm8, 32);
5 index = (P == '1'); add = (U == '1'); wback = (W == '1');
6 if t IN {13,15} || (wback && n == t) then UNPREDICTABLE;

```

### CONSTRAINED UNPREDICTABLE behavior

If `wback && n == t`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction executes but the value stored is UNKNOWN.

### Assembler symbols for all encodings

|       |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                  |
|-------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <c>   | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |
| <q>   | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |
| <Rt>  | Is the general-purpose register to be transferred, encoded in the "Rt" field.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                    |
| <Rn>  | Is the general-purpose base register, encoded in the "Rn" field.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                 |
| +/-   | Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in the "U" field. It can have the following values: <ul style="list-style-type: none"> <li>- when U = 0</li> <li>+ when U = 1</li> </ul>                                                                                                                                                                                                                                                                                                                                                                           |
| +     | Specifies the offset is added to the base register.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                              |
| <imm> | For the post-indexed or pre-indexed variant: is an 8-bit unsigned immediate byte offset, in the range 0 to 255, encoded in the "imm8" field.<br>For encoding T1: is the optional positive unsigned immediate byte offset, a multiple of 2 in the range 0 to 62, defaulting to 0 and encoded in the "imm5" field as <imm>/2.<br>For encoding T2: is an optional 12-bit unsigned immediate byte offset, in the range 0 to 4095, defaulting to 0 and encoded in the "imm12" field.<br>For encoding T3: is an optional 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 and encoded in the "imm8" field. |

### Operation for all encodings

```

1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 offset_addr = if add then (R[n] + imm32) else (R[n] - imm32);
4 address = if index then offset_addr else R[n];
5
6 // Determine if the stack pointer limit should be checked
7 if n == 13 && wback then
8 (limit, applylimit) = LookUpSPLim(LookUpSP());
9 else
10 applylimit = FALSE;
11 // Memory operation only performed if limit not violated
12 if !applylimit || (UInt(offset_addr) >= UInt(limit)) then

```

```
13 MemU[address,2] = R[t][15:0];
14
15 // If the stack pointer is being updated a fault will be raised if
16 // the limit is violated
17 if wback then RSPCheck[n] = offset_addr;
```

### C2.4.196 STRH (register)

Store Register Halfword (register). Store Register Halfword (register) calculates an address from a base register value and an offset register value, and stores a halfword from a register to memory. The offset register value can be shifted left by 0, 1, 2, or 3 bits.

#### T1

Armv8-M

|    |    |    |    |    |    |   |   |    |   |    |   |   |   |   |    |
|----|----|----|----|----|----|---|---|----|---|----|---|---|---|---|----|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7  | 6 | 5  | 4 | 3 | 2 | 1 | 0  |
| 0  | 1  | 0  | 1  | 0  | 0  | 1 |   | Rm |   | Rn |   |   |   |   | Rt |

#### T1 variant

STRH{<c>}{<q>} <Rt>, [<Rn>, {+}<Rm>]

#### Decode for this encoding

```
1 t = UInt(Rt); n = UInt(Rn); m = UInt(Rm);
2 index = TRUE; add = TRUE; wback = FALSE;
3 (shift_t, shift_n) = (SRTYPE_LSL, 0);
```

#### T2

Armv8-M Main Extension only

|    |    |    |    |    |    |   |   |   |   |   |            |   |    |   |   |    |    |    |    |    |      |   |   |    |   |   |   |   |   |   |   |
|----|----|----|----|----|----|---|---|---|---|---|------------|---|----|---|---|----|----|----|----|----|------|---|---|----|---|---|---|---|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4          | 3 | 2  | 1 | 0 | 15 | 14 | 13 | 12 | 11 | 10   | 9 | 8 | 7  | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 1  | 1  | 1  | 1  | 1  | 0  | 0 | 0 | 0 | 1 | 0 | Rn != 1111 |   | Rt |   | 0 | 0  | 0  | 0  | 0  | 0  | imm2 |   |   | Rm |   |   |   |   |   |   |   |

#### T2 variant

STRH{<c>}.W <Rt>, [<Rn>, {+}<Rm>]  
 // <Rt>, <Rn>, <Rm> can be represented in T1  
 STRH{<c>}{<q>} <Rt>, [<Rn>, {+}<Rm>{, LSL #<imm>}]

#### Decode for this encoding

```
1 if Rn == '1111' then UNDEFINED;
2 if !HaveMainExt() then UNDEFINED;
3 t = UInt(Rt); n = UInt(Rn); m = UInt(Rm);
4 index = TRUE; add = TRUE; wback = FALSE;
5 (shift_t, shift_n) = (SRTYPE_LSL, UInt(imm2));
6 if t IN {13,15} || m IN {13,15} then UNPREDICTABLE;
```

#### Assembler symbols for all encodings

|       |                                                                                                                                                                                   |
|-------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <c>   | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                                                                                                          |
| <q>   | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                                                                                                          |
| <Rt>  | Is the general-purpose register to be transferred, encoded in the "Rt" field.                                                                                                     |
| <Rn>  | Is the general-purpose base register, encoded in the "Rn" field.                                                                                                                  |
| +     | Specifies the index register is added to the base register.                                                                                                                       |
| <Rm>  | Is the general-purpose index register, encoded in the "Rm" field.                                                                                                                 |
| <imm> | If present, the size of the left shift to apply to the value from <Rm>, in the range 1-3. <imm> is encoded in imm2. If absent, no shift is specified and imm2 is encoded as 0b00. |

#### Operation for all encodings

```
1 if ConditionPassed() then
```

## Chapter C2. Instruction Specification

### C2.4. Alphabetical list of instructions

```
2 EncodingSpecificOperations();
3 offset = Shift(R[m], shift_t, shift_n, APSR.C);
4 address = R[n] + offset;
5 MemU[address,2] = R[t][15:0];
```

## C2.4.197 STRHT

Store Register Halfword Unprivileged. Store Register Halfword Unprivileged calculates an address from a base register value and an immediate offset, and stores a halfword from a register to memory.

When privileged software uses an **STRHT** instruction, the memory access is restricted as if the software was unprivileged.

### T1

*Armv8-M Main Extension only*

|    |    |    |    |    |    |   |   |   |   |   |   |            |   |   |   |    |    |    |    |    |    |   |   |      |   |   |   |   |   |   |   |
|----|----|----|----|----|----|---|---|---|---|---|---|------------|---|---|---|----|----|----|----|----|----|---|---|------|---|---|---|---|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3          | 2 | 1 | 0 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7    | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 1  | 1  | 1  | 1  | 1  | 0  | 0 | 0 | 0 | 0 | 1 | 0 | Rn != 1111 |   |   |   | Rt |    |    |    | 1  | 1  | 1 | 0 | imm8 |   |   |   |   |   |   |   |

### T1 variant

STRHT{<c>}{<q>} <Rt>, [<Rn> {, #<+><imm>}]

### Decode for this encoding

```

1 if Rn == '1111' then UNDEFINED;
2 if !HaveMainExt() then UNDEFINED;
3 t = UInt(Rt); n = UInt(Rn); postindex = FALSE; add = TRUE;
4 register_form = FALSE; imm32 = ZeroExtend(imm8, 32);
5 if t IN {13,15} then UNPREDICTABLE;

```

### Assembler symbols for all encodings

|       |                                                                                                                              |
|-------|------------------------------------------------------------------------------------------------------------------------------|
| <c>   | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                                                     |
| <q>   | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                                                     |
| <Rt>  | Is the general-purpose register to be transferred, encoded in the "Rt" field.                                                |
| <Rn>  | Is the general-purpose base register, encoded in the "Rn" field.                                                             |
| +     | Specifies the offset is added to the base register.                                                                          |
| <imm> | Is an optional 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 and encoded in the "imm8" field. |

### Operation for all encodings

```

1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 address = R[n] + imm32;
4 MemU_unpriv[address,2] = R[t][15:0];

```



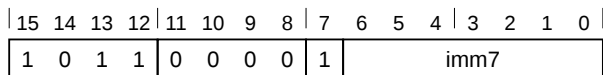


### C2.4.199 SUB (SP minus immediate)

Subtract from SP (immediate). Subtract (SP minus immediate) subtracts an immediate value from the SP value, and writes the result to the destination register.

#### T1

Armv8-M



#### T1 variant

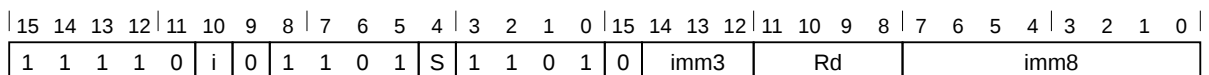
SUB{<c>}{<q>} {SP,} SP, #<imm7>

#### Decode for this encoding

```
1 d = 13; setflags = FALSE; imm32 = ZeroExtend(imm7:'00', 32);
```

#### T2

Armv8-M Main Extension only



#### SUB variant

Applies when S == 0.

SUB{<c>}.W {<Rd>,} SP, #<const>  
 // <Rd>, <const> can be represented in T1  
 SUB{<c>}{<q>} {<Rd>,} SP, #<const>

#### SUBS variant

Applies when S == 1 && Rd != 1111.

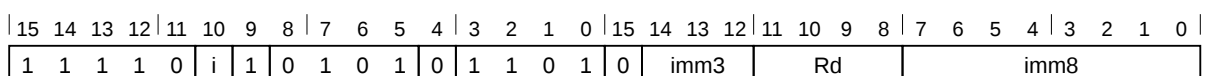
SUBS{<c>}{<q>} {<Rd>,} SP, #<const>

#### Decode for this encoding

```
1 if Rd == '1111' && S == '1' then SEE "CMP (immediate)";
2 if !HaveMainExt() then UNDEFINED;
3 d = UInt(Rd); setflags = (S == '1'); imm32 = T32ExpandImm(i:imm3:imm8);
4 if d == 15 && S == '0' then UNPREDICTABLE;
```

#### T3

Armv8-M Main Extension only



#### T3 variant

```

SUB{<c>}{<q>} {<Rd>}, SP, #<imm12>
 // <imm12> cannot be represented in T1, T2, or T3
SUBW{<c>}{<q>} {<Rd>}, SP, #<imm12>
 // <imm12> can be represented in T1, T2, or T3

```

### Decode for this encoding

```

1 if !HaveMainExt() then UNDEFINED;
2 d = UInt(Rd); setflags = FALSE; imm32 = ZeroExtend(i:imm3:imm8, 32);
3 if d == 15 then UNPREDICTABLE;

```

### Assembler symbols for all encodings

|         |                                                                                                                                                                                                |
|---------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <c>     | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                                                                                                                       |
| <q>     | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                                                                                                                       |
| <imm7>  | Is an unsigned immediate, a multiple of 4 in the range 0 to 508, encoded in the "imm7" field as <imm7>/4.                                                                                      |
| <Rd>    | Is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the SP.                                                                                   |
| <imm12> | Is a 12-bit unsigned immediate, in the range 0 to 4095, encoded in the "i:imm3:imm8" field.                                                                                                    |
| <const> | Is an immediate value derived from the 12-bit immediate that is encoded in the 'i:imm3:imm8' field. See <a href="#">C1.5 Modified immediate constants</a> on page 324 for the range of values. |

### Operation for all encodings

```

1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 (result, carry, overflow) = AddWithCarry(SP, NOT(imm32), '1');
4 RSPCheck[d] = result;
5 if setflags then
6 APSR.N = result[31];
7 APSR.Z = IsZeroBit(result);
8 APSR.C = carry;
9 APSR.V = overflow;

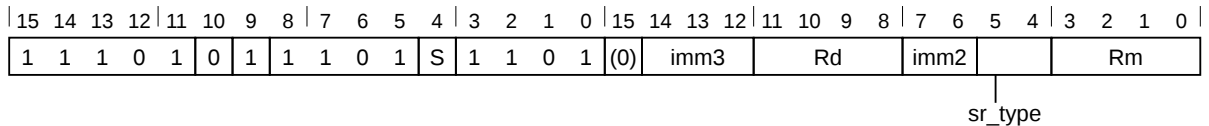
```

## C2.4.200 SUB (SP minus register)

Subtract from SP (register). Subtract (SP minus register) subtracts an optionally-shifted register value from the SP value, and writes the result to the destination register.

### T1

Armv8-M Main Extension only



### SUB, rotate right with extend variant

Applies when **S == 0** && **imm3 == 000** && **imm2 == 00** && **sr\_type == 11**.

SUB{<c>}{<q>} {<Rd>}, SP, <Rm>, RRX

### SUB, shift or rotate by value variant

Applies when **S == 0** && **!(imm3 == 000 && imm2 == 00 && sr\_type == 11)**.

SUB{<c>}.W {<Rd>}, SP, <Rm>

// <Rd>, <Rm> can be represented in T1 or T2

SUB{<c>}{<q>} {<Rd>}, SP, <Rm> {, <shift> #<amount>}

### SUBS, rotate right with extend variant

Applies when **S == 1** && **imm3 == 000** && **Rd != 1111** && **imm2 == 00** && **sr\_type == 11**.

SUBS{<c>}{<q>} {<Rd>}, SP, <Rm>, RRX

### SUBS, shift or rotate by value variant

Applies when **S == 1** && **!(imm3 == 000 && imm2 == 00 && sr\_type == 11)** && **Rd != 1111**.

SUBS{<c>}{<q>} {<Rd>}, SP, <Rm> {, <shift> #<amount>}

### Decode for this encoding

```

1 if Rd == '1111' && S == '1' then SEE "CMP (register)";
2 if !HaveMainExt() then UNDEFINED;
3 d = UInt(Rd); m = UInt(Rm); setflags = (S == '1');
4 (shift_t, shift_n) = DecodeImmShift(sr_type, imm3:imm2);
5 if d == 13 && (shift_t != SRTYPE_LSL || shift_n > 3) then UNPREDICTABLE;
6 if (d == 15 && S == '0') || m IN {13,15} then UNPREDICTABLE;
```

### Assembler symbols for all encodings

<c> See [C1.2.5 Standard assembler syntax fields](#) on page 309.  
 <q> See [C1.2.5 Standard assembler syntax fields](#) on page 309.  
 <Rd> Is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the SP.  
 <Rm> Is the second general-purpose source register, encoded in the "Rm" field.  
 <shift> Is the type of shift to be applied to the second source register, encoded in the "sr\_type" field. It can have the following values:

LSL when `sr_type = 00`  
 LSR when `sr_type = 01`  
 ASR when `sr_type = 10`  
 ROR when `sr_type = 11`

`<amount>` Is the shift amount, in the range 1 to 31 (when `<shift> = LSL` or `ROR`) or 1 to 32 (when `<shift> = LSR` or `ASR`) encoded in the "imm3:imm2" field as `<amount> modulo 32`.

### Operation for all encodings

```

1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 shifted = Shift(R[m], shift_t, shift_n, APSR.C);
4 (result, carry, overflow) = AddWithCarry(SP, NOT(shifted), '1');
5 RSPCheck[d] = result;
6 if setflags then
7 APSR.N = result[31];
8 APSR.Z = IsZeroBit(result);
9 APSR.C = carry;
10 APSR.V = overflow;

```

### C2.4.201 SUB (immediate)

Subtract (immediate). Subtract (immediate) subtracts an immediate value from a register value, and writes the result to the destination register. It can optionally update the condition flags based on the result.

#### T1

Armv8-M

|    |    |    |    |    |    |   |      |   |   |   |    |   |    |   |   |
|----|----|----|----|----|----|---|------|---|---|---|----|---|----|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8    | 7 | 6 | 5 | 4  | 3 | 2  | 1 | 0 |
| 0  | 0  | 0  | 1  | 1  | 1  | 1 | imm3 |   |   |   | Rn |   | Rd |   |   |

#### T1 variant

```
SUB<c>{<q>} <Rd>, <Rn>, #<imm3>
 // Inside IT block
SUBS{<q>} <Rd>, <Rn>, #<imm3>
 // Outside IT block
```

#### Decode for this encoding

```
1 d = UInt(Rd); n = UInt(Rn); setflags = !InITBlock(); imm32 = ZeroExtend(imm3, 32);
```

#### T2

Armv8-M

|    |    |    |    |    |     |   |   |      |   |   |   |   |   |   |   |
|----|----|----|----|----|-----|---|---|------|---|---|---|---|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10  | 9 | 8 | 7    | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 0  | 0  | 1  | 1  | 1  | Rdn |   |   | imm8 |   |   |   |   |   |   |   |

#### T2 variant

```
SUB<c>{<q>} <Rdn>, #<imm8>
 // Inside IT block, and <Rdn>, <imm8> can be represented in T1
SUB<c>{<q>} {<Rdn>, } <Rdn>, #<imm8>
 // Inside IT block, and <Rdn>, <imm8> cannot be represented in T1
SUBS{<q>} <Rdn>, #<imm8>
 // Outside IT block, and <Rdn>, <imm8> can be represented in T1
SUBS{<q>} {<Rdn>, } <Rdn>, #<imm8>
 // Outside IT block, and <Rdn>, <imm8> cannot be represented in T1
```

#### Decode for this encoding

```
1 d = UInt(Rdn); n = UInt(Rdn); setflags = !InITBlock(); imm32 = ZeroExtend(imm8, 32);
```

#### T3

Armv8-M Main Extension only

|    |    |    |    |    |    |   |   |   |   |   |   |            |   |      |   |    |    |      |    |    |    |   |   |   |   |   |   |   |   |   |   |
|----|----|----|----|----|----|---|---|---|---|---|---|------------|---|------|---|----|----|------|----|----|----|---|---|---|---|---|---|---|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3          | 2 | 1    | 0 | 15 | 14 | 13   | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 1  | 1  | 1  | 1  | 0  | i  | 0 | 1 | 1 | 0 | 1 | S | Rn != 1101 | 0 | imm3 |   | Rd |    | imm8 |    |    |    |   |   |   |   |   |   |   |   |   |   |

#### SUB variant

Applies when S == 0.

```
SUB<c>.W {<Rd>, } <Rn>, #<const>
 // Inside IT block, and <Rd>, <Rn>, <const> can be represented in T1 or T2
SUB{<c>}{<q>} {<Rd>, } <Rn>, #<const>
```

### SUBS variant

Applies when **S == 1 && Rd != 1111**.

```
SUBS.W {<Rd>}, {<Rn>, #<const>}
// Outside IT block, and <Rd>, <Rn>, <const> can be represented in T1 or T2
SUBS{<c>}{<q>} {<Rd>}, {<Rn>, #<const>}
```

### Decode for this encoding

```
1 if Rd == '1111' && S == '1' then SEE "CMP (immediate)";
2 if Rn == '1101' then SEE "SUB (SP minus immediate)"
3 if !HaveMainExt() then UNDEFINED;
4 d = UInt(Rd); n = UInt(Rn); setflags = (S == '1'); imm32 = T32ExpandImm(i:imm3:imm8);
5 if d == 13 || (d == 15 && S == '0') || n == 15 then UNPREDICTABLE;
```

### T4

Armv8-M Main Extension only

|    |    |    |    |    |    |   |   |   |   |   |   |            |   |      |    |      |    |    |    |    |    |   |   |   |   |   |   |   |   |   |   |
|----|----|----|----|----|----|---|---|---|---|---|---|------------|---|------|----|------|----|----|----|----|----|---|---|---|---|---|---|---|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3          | 2 | 1    | 0  | 15   | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 1  | 1  | 1  | 1  | 0  | i  | 1 | 0 | 1 | 0 | 1 | 0 | Rn != 11x1 | 0 | imm3 | Rd | imm8 |    |    |    |    |    |   |   |   |   |   |   |   |   |   |   |

### T4 variant

```
SUB{<c>}{<q>} {<Rd>}, {<Rn>, #<imm12>}
// <imm12> cannot be represented in T1, T2, or T3
SUBW{<c>}{<q>} {<Rd>}, {<Rn>, #<imm12>}
// <imm12> can be represented in T1, T2, or T3
```

### Decode for this encoding

```
1 if Rn == '1111' then SEE ADR;
2 if Rn == '1101' then SEE "SUB (SP minus immediate)"
3 if !HaveMainExt() then UNDEFINED;
4 d = UInt(Rd); n = UInt(Rn); setflags = FALSE; imm32 = ZeroExtend(i:imm3:imm8, 32);
5 if d IN {13,15} then UNPREDICTABLE;
```

### Assembler symbols for all encodings

- <c> See [C1.2.5 Standard assembler syntax fields](#) on page 309.
- <q> See [C1.2.5 Standard assembler syntax fields](#) on page 309.
- <Rdn> Is the general-purpose source and destination register, encoded in the "Rdn" field.
- <imm8> Is an 8-bit unsigned immediate, in the range 0 to 255, encoded in the "imm8" field.
- <Rd> Is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as <Rn>.
- <Rn> For encoding T1: is the general-purpose source register, encoded in the "Rn" field.  
 For encoding T3: is the general-purpose source register, encoded in the "Rn" field. If the SP is used, see [C2.4.199 SUB \(SP minus immediate\)](#) on page 685.  
 For encoding T4: is the general-purpose source register, encoded in the "Rn" field. If the SP is used, see [C2.4.199 SUB \(SP minus immediate\)](#) on page 685. If the PC is used, see [C2.4.8 ADR](#) on page 383.
- <imm3> Is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the "imm3" field.
- <imm12> Is a 12-bit unsigned immediate, in the range 0 to 4095, encoded in the "i:imm3:imm8" field.
- <const> Is an immediate value derived from the 12-bit immediate that is encoded in the 'i:imm3:imm8' field. See [C1.5 Modified immediate constants](#) on page 324 for the range of values.

### Operation for all encodings

```
1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 (result, carry, overflow) = AddWithCarry(R[n], NOT(imm32), '1');
4 R[d] = result;
5 if setflags then
6 APSR.N = result[31];
7 APSR.Z = IsZeroBit(result);
8 APSR.C = carry;
9 APSR.V = overflow;
```

## C2.4.202 SUB (immediate, from PC)

Subtract from PC. Subtract from PC subtracts an immediate value from the `Align(PC, 4)` value to form a PC-relative address, and writes the result to the destination register. Arm recommends that, where possible, software avoids using this alias.

This instruction is an alias of the [ADR](#) instruction. This means that:

- The encodings in this description are named to match the encodings of [ADR](#).
- The description of [ADR](#) gives the operational pseudocode for this instruction.

### T2

*Armv8-M Main Extension only*

|    |    |    |    |    |    |   |   |   |   |   |   |   |   |   |   |    |      |    |    |    |    |   |   |   |      |   |   |   |   |   |   |  |
|----|----|----|----|----|----|---|---|---|---|---|---|---|---|---|---|----|------|----|----|----|----|---|---|---|------|---|---|---|---|---|---|--|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | 15 | 14   | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6    | 5 | 4 | 3 | 2 | 1 | 0 |  |
| 1  | 1  | 1  | 1  | 0  | i  | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 1 | 1 | 1 | 0  | imm3 |    |    |    | Rd |   |   |   | imm8 |   |   |   |   |   |   |  |

### T2 variant

`SUB{<c>}{<q>} <Rd>, PC, #<imm12>`

### is equivalent to

`ADR{<c>}{<q>} <Rd>, <label>`

and is the preferred disassembly when `i:imm3:imm8 == '000000000000'`.

### Assembler symbols for all encodings

|                            |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                              |
|----------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>&lt;c&gt;</code>     | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                     |
| <code>&lt;q&gt;</code>     | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                     |
| <code>&lt;Rd&gt;</code>    | Is the general-purpose destination register, encoded in the "Rd" field.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                      |
| <code>&lt;label&gt;</code> | For encoding T1: the label of an instruction or literal data item whose address is to be loaded into <code>&lt;Rd&gt;</code> . The assembler calculates the required value of the offset from the <code>Align(PC, 4)</code> value of the <code>ADR</code> instruction to this label. Permitted values of the size of the offset are multiples of 4 in the range 0 to 1020.<br>For encoding T2 and T3: the label of an instruction or literal data item whose address is to be loaded into <code>&lt;Rd&gt;</code> . The assembler calculates the required value of the offset from the <code>Align(PC, 4)</code> value of the <code>ADR</code> instruction to this label. If the offset is zero or positive, encoding T3 is used, with <code>imm32</code> equal to the offset. If the offset is negative, encoding T2 is used, with <code>imm32</code> equal to the size of the offset. That is, the use of encoding T2 indicates that the required offset is minus the value of <code>imm32</code> . Permitted values of the size of the offset are 0-4095. |
| <code>&lt;imm12&gt;</code> | Is a 12-bit unsigned immediate, in the range 0 to 4095, encoded in the "i:imm3:imm8" field.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                  |

### Operation for all encodings

The description of [ADR](#) gives the operational pseudocode for this instruction.

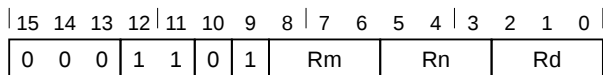


### C2.4.203 SUB (register)

Subtract (register). Subtract (register) subtracts an optionally-shifted register value from a register value, and writes the result to the destination register. It can optionally update the condition flags based on the result.

#### T1

Armv8-M



#### T1 variant

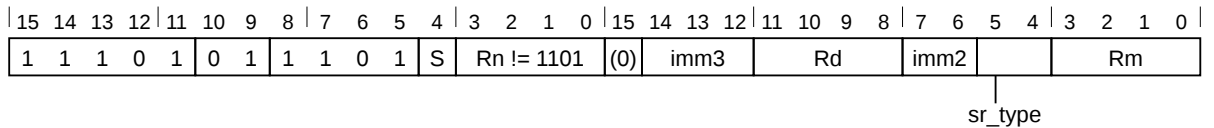
```
SUB<c>{<q>} <Rd>, <Rn>, <Rm>
 // Inside IT block
SUBS{<q>} {<Rd>, } <Rn>, <Rm>
 // Outside IT block
```

#### Decode for this encoding

```
1 d = UInt(Rd); n = UInt(Rn); m = UInt(Rm); setflags = !InITBlock();
2 (shift_t, shift_n) = (SRTType_LSL, 0);
```

#### T2

Armv8-M Main Extension only



#### SUB, rotate right with extend variant

Applies when **S == 0** && **imm3 == 000** && **imm2 == 00** && **sr\_type == 11**.

```
SUB{<c>}{<q>} {<Rd>, } <Rn>, <Rm>, RRX
```

#### SUB, shift or rotate by value variant

Applies when **S == 0** && **!(imm3 == 000 && imm2 == 00 && sr\_type == 11)**.

```
SUB<c>.W {<Rd>, } <Rn>, <Rm>
 // Inside IT block, and <Rd>, <Rn>, <Rm> can be represented in T1
SUB{<c>}{<q>} {<Rd>, } <Rn>, <Rm> {, <shift> #<amount>}
```

#### SUBS, rotate right with extend variant

Applies when **S == 1** && **imm3 == 000** && **Rd != 1111** && **imm2 == 00** && **sr\_type == 11**.

```
SUBS{<c>}{<q>} {<Rd>, } <Rn>, <Rm>, RRX
```

#### SUBS, shift or rotate by value variant

Applies when **S == 1** && **!(imm3 == 000 && imm2 == 00 && sr\_type == 11) && Rd != 1111**.

```

SUBS.W {<Rd>,} <Rn>, <Rm>
 // Outside IT block, and <Rd>, <Rn>, <Rm> can be represented in T1
SUBS{<c>}{<q>} {<Rd>,} <Rn>, <Rm> {, <shift> #<amount>}

```

### Decode for this encoding

```

1 if Rd == '1111' && S == '1' then SEE "CMP (register)";
2 if Rn == '1101' then SEE "SUB (SP minus register)"
3 if !HaveMainExt() then UNDEFINED;
4 d = UInt(Rd); n = UInt(Rn); m = UInt(Rm); setflags = (S == '1');
5 (shift_t, shift_n) = DecodeImmShift(sr_type, imm3:imm2);
6 if d == 13 || (d == 15 && S == '0') || n == 15 || m IN {13,15} then UNPREDICTABLE;

```

### Assembler symbols for all encodings

<c> See [C1.2.5 Standard assembler syntax fields](#) on page 309.

<q> See [C1.2.5 Standard assembler syntax fields](#) on page 309.

<Rd> Is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as <Rn>.

<Rn> For encoding T1: is the first general-purpose source register, encoded in the "Rn" field. For encoding T2: is the first general-purpose source register, encoded in the "Rn" field. If the SP is used, see [C2.4.200 SUB \(SP minus register\)](#) on page 687.

<Rm> Is the second general-purpose source register, encoded in the "Rm" field.

<shift> Is the type of shift to be applied to the second source register, encoded in the "sr\_type" field. It can have the following values:

|     |                   |
|-----|-------------------|
| LSL | when sr_type = 00 |
| LSR | when sr_type = 01 |
| ASR | when sr_type = 10 |
| ROR | when sr_type = 11 |

<amount> Is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR) encoded in the "imm3:imm2" field as <amount> modulo 32.

### Operation for all encodings

```

1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 shifted = Shift(R[m], shift_t, shift_n, APSR.C);
4 (result, carry, overflow) = AddWithCarry(R[n], NOT(shifted), '1');
5 R[d] = result;
6 if setflags then
7 APSR.N = result[31];
8 APSR.Z = IsZeroBit(result);
9 APSR.C = carry;
10 APSR.V = overflow;

```

## C2.4.204 SVC

Supervisor Call. The Supervisor Call instruction generates a call to a system supervisor.

Use it as a call to an operating system to provide a service.

In older versions of the Arm architecture, SVC was called SWI, Software Interrupt.

### T1

Armv8-M

|    |    |    |    |    |    |   |   |      |   |   |   |   |   |   |   |
|----|----|----|----|----|----|---|---|------|---|---|---|---|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7    | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 1  | 1  | 0  | 1  | 1  | 1  | 1 | 1 | imm8 |   |   |   |   |   |   |   |

### T1 variant

SVC{<c>}{<q>} {#}<imm>

### Decode for this encoding

```
1 imm32 = ZeroExtend(imm8, 32);
2 // imm32 is for assembly/disassembly. SVC handlers in some
3 // systems interpret imm8 in software, for example to determine the required service.
```

### Assembler symbols for all encodings

<c> See [C1.2.5 Standard assembler syntax fields](#) on page 309.  
<q> See [C1.2.5 Standard assembler syntax fields](#) on page 309.  
<imm> Is an 8-bit unsigned immediate, in the range 0 to 255, encoded in the "imm8" field.

### Operation for all encodings

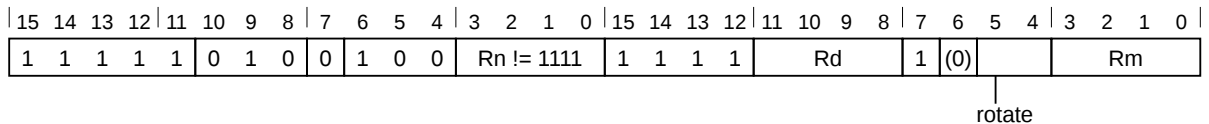
```
1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 CallSupervisor();
```

## C2.4.205 SXTAB

Signed Extend and Add Byte. Signed Extend and Add Byte extracts an 8-bit value from a register, sign-extends it to 32 bits, adds the result to the value in another register, and writes the final result to the destination register. The instruction can specify a rotation by 0, 8, 16, or 24 bits before extracting the 8-bit value.

### T1

Armv8-M DSP Extension only



### T1 variant

SXTAB{<c>}{<q>} {<Rd>}, <Rn>, <Rm> {, ROR #<amount>}

### Decode for this encoding

```

1 if Rn == '1111' then SEE SXTB;
2 if !HaveDSPExt() then UNDEFINED;
3 d = UInt(Rd); n = UInt(Rn); m = UInt(Rm); rotation = UInt(rotate:'000');
4 if d IN {13,15} || n == 13 || m IN {13,15} then UNPREDICTABLE;

```

### Assembler symbols for all encodings

<c> See [C1.2.5 Standard assembler syntax fields](#) on page 309.  
 <q> See [C1.2.5 Standard assembler syntax fields](#) on page 309.  
 <Rd> Is the general-purpose destination register, encoded in the "Rd" field.  
 <Rn> Is the first general-purpose source register, encoded in the "Rn" field.  
 <Rm> Is the second general-purpose source register, encoded in the "Rm" field.  
 <amount> Is the rotate amount, encoded in the "rotate" field. It can have the following values:  
     0 when rotate = 00  
     8 when rotate = 01  
     16 when rotate = 10  
     24 when rotate = 11  
 ROR #<amount> can be omitted, meaning a rotate amount of 0. This is the preferred disassembly for rotate == 0b00.

### Operation for all encodings

```

1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 rotated = ROR(R[m], rotation);
4 R[d] = R[n] + SignExtend(rotated[7:0], 32);

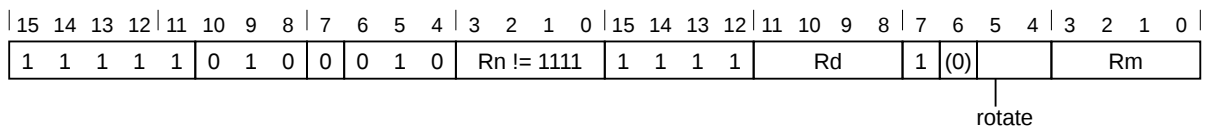
```

## C2.4.206 SXTAB16

Signed Extend and Add Byte 16. Signed Extend and Add Byte 16 extracts two 8-bit values from a register, sign-extends them to 16 bits each, adds the results to two 16-bit values from another register, and writes the final results to the destination register. The instruction can specify a rotation by 0, 8, 16, or 24 bits before extracting the 8-bit values.

### T1

Armv8-M DSP Extension only



### T1 variant

SXTAB16{<c>}{<q>} {<Rd>, } <Rn>, <Rm> {, ROR #<amount>}

### Decode for this encoding

```

1 if Rn == '1111' then SEE SXTB16;
2 if !HaveDSPExt() then UNDEFINED;
3 d = UInt(Rd); n = UInt(Rn); m = UInt(Rm); rotation = UInt(rotate:'000');
4 if d IN {13,15} || n == 13 || m IN {13,15} then UNPREDICTABLE;

```

### Assembler symbols for all encodings

|          |                                                                                                                                                                                                                                                                                                                                                                                                                                                          |   |                  |   |                  |    |                  |    |                  |
|----------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---|------------------|---|------------------|----|------------------|----|------------------|
| <c>      | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                                                                                                                                                                                                                                                                                                                                                                                 |   |                  |   |                  |    |                  |    |                  |
| <q>      | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                                                                                                                                                                                                                                                                                                                                                                                 |   |                  |   |                  |    |                  |    |                  |
| <Rd>     | Is the general-purpose destination register, encoded in the "Rd" field.                                                                                                                                                                                                                                                                                                                                                                                  |   |                  |   |                  |    |                  |    |                  |
| <Rn>     | Is the first general-purpose source register, encoded in the "Rn" field.                                                                                                                                                                                                                                                                                                                                                                                 |   |                  |   |                  |    |                  |    |                  |
| <Rm>     | Is the second general-purpose source register, encoded in the "Rm" field.                                                                                                                                                                                                                                                                                                                                                                                |   |                  |   |                  |    |                  |    |                  |
| <amount> | Is the rotate amount, encoded in the "rotate" field. It can have the following values: <table style="margin-left: 20px; border: none;"> <tr><td>0</td><td>when rotate = 00</td></tr> <tr><td>8</td><td>when rotate = 01</td></tr> <tr><td>16</td><td>when rotate = 10</td></tr> <tr><td>24</td><td>when rotate = 11</td></tr> </table> ROR #<amount> can be omitted, meaning a rotate amount of 0. This is the preferred disassembly for rotate == 0b00. | 0 | when rotate = 00 | 8 | when rotate = 01 | 16 | when rotate = 10 | 24 | when rotate = 11 |
| 0        | when rotate = 00                                                                                                                                                                                                                                                                                                                                                                                                                                         |   |                  |   |                  |    |                  |    |                  |
| 8        | when rotate = 01                                                                                                                                                                                                                                                                                                                                                                                                                                         |   |                  |   |                  |    |                  |    |                  |
| 16       | when rotate = 10                                                                                                                                                                                                                                                                                                                                                                                                                                         |   |                  |   |                  |    |                  |    |                  |
| 24       | when rotate = 11                                                                                                                                                                                                                                                                                                                                                                                                                                         |   |                  |   |                  |    |                  |    |                  |

### Operation for all encodings

```

1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 rotated = ROR(R[m], rotation);
4 bits(32) result;
5 result[15:0] = R[n][15:0] + SignExtend(rotated[7:0], 16);
6 result[31:16] = R[n][31:16] + SignExtend(rotated[23:16], 16);
7 R[d] = result;

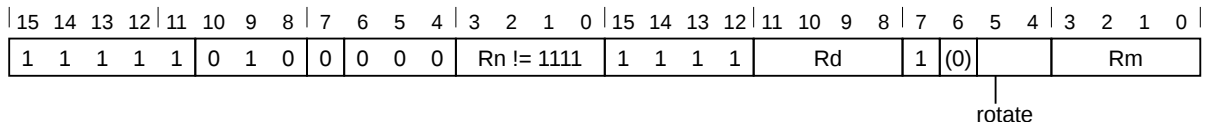
```

## C2.4.207 SXTAH

Signed Extend and Add Halfword. Signed Extend and Add Halfword extracts a 16-bit value from a register, sign-extends it to 32 bits, adds the result to a value from another register, and writes the final result to the destination register. The instruction can specify a rotation by 0, 8, 16, or 24 bits before extracting the 16-bit value.

### T1

Armv8-M DSP Extension only



### T1 variant

SXTAH<c>{<q>} {<Rd>,<Rn>,<Rm> {,<ROR #<amount>}}

### Decode for this encoding

```

1 if Rn == '1111' then SEE SXTAH;
2 if !HaveDSPExt() then UNDEFINED;
3 d = UInt(Rd); n = UInt(Rn); m = UInt(Rm); rotation = UInt(rotate:'000');
4 if d IN {13,15} || n == 13 || m IN {13,15} then UNPREDICTABLE;

```

### Assembler symbols for all encodings

- <c> See [C1.2.5 Standard assembler syntax fields](#) on page 309.
  - <q> See [C1.2.5 Standard assembler syntax fields](#) on page 309.
  - <Rd> Is the general-purpose destination register, encoded in the "Rd" field.
  - <Rn> Is the first general-purpose source register, encoded in the "Rn" field.
  - <Rm> Is the second general-purpose source register, encoded in the "Rm" field.
  - <amount> Is the rotate amount, encoded in the "rotate" field. It can have the following values:
    - 0 when rotate = 00
    - 8 when rotate = 01
    - 16 when rotate = 10
    - 24 when rotate = 11
- ROR #<amount> can be omitted, meaning a rotate amount of 0. This is the preferred disassembly for rotate == 0b00.

### Operation for all encodings

```

1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 rotated = ROR(R[m], rotation);
4 R[d] = R[n] + SignExtend(rotated[15:0], 32);

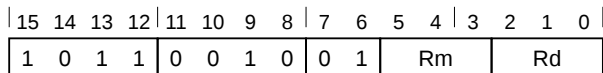
```

## C2.4.208 SXTB

Signed Extend Byte. Signed Extend Byte extracts an 8-bit value from a register, sign extends it to 32 bits, and writes the result to the destination register. The instruction can specify a rotation by 0, 8, 16, or 24 bits before extracting the 8-bit value.

### T1

Armv8-M



### T1 variant

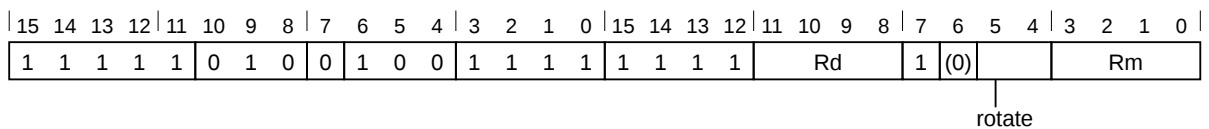
SXTB{<c>}{<q>} {<Rd>}, {<Rm>}

### Decode for this encoding

```
1 d = UInt(Rd); m = UInt(Rm); rotation = 0;
```

### T2

Armv8-M Main Extension only



### T2 variant

SXTB{<c>}.W {<Rd>}, {<Rm>}  
 // <Rd>, <Rm> can be represented in T1  
 SXTB{<c>}{<q>} {<Rd>}, {<Rm>} {, ROR #<amount>}

### Decode for this encoding

```
1 if !HaveMainExt() then UNDEFINED;
2 d = UInt(Rd); m = UInt(Rm); rotation = UInt(rotate: '000');
3 if d IN {13,15} || m IN {13,15} then UNPREDICTABLE;
```

### Assembler symbols for all encodings

- <c> See [C1.2.5 Standard assembler syntax fields](#) on page 309.
  - <q> See [C1.2.5 Standard assembler syntax fields](#) on page 309.
  - <Rd> Is the general-purpose destination register, encoded in the "Rd" field.
  - <Rm> Is the general-purpose source register, encoded in the "Rm" field.
  - <amount> Is the rotate amount, encoded in the "rotate" field. It can have the following values:
    - 0 when rotate = 00
    - 8 when rotate = 01
    - 16 when rotate = 10
    - 24 when rotate = 11
- ROR #<amount> can be omitted, meaning a rotate amount of 0. This is the preferred disassembly for rotate == 0b00.

### Operation for all encodings

```
1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 rotated = ROR(R[m], rotation);
4 R[d] = SignExtend(rotated[7:0], 32);
```

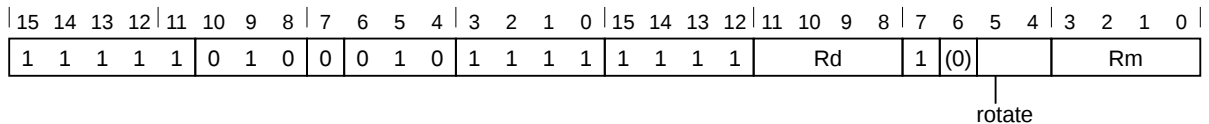


## C2.4.209 SXTB16

Signed Extend Byte 16. Signed Extend Byte 16 extracts two 8-bit values from a register, sign-extends them to 16 bits each, and writes the results to the destination register. The instruction can specify a rotation by 0, 8, 16, or 24 bits before extracting the 8-bit values.

### T1

Armv8-M DSP Extension only



### T1 variant

SXTB16{<c>}{<q>} {<Rd>,<Rm> {,<ROR #<amount>}}

### Decode for this encoding

```

1 if !HaveDSPExt() then UNDEFINED;
2 d = UInt(Rd); m = UInt(Rm); rotation = UInt(rotate:'000');
3 if d IN {13,15} || m IN {13,15} then UNPREDICTABLE;

```

### Assembler symbols for all encodings

- <c> See [C1.2.5 Standard assembler syntax fields](#) on page 309.
- <q> See [C1.2.5 Standard assembler syntax fields](#) on page 309.
- <Rd> Is the general-purpose destination register, encoded in the "Rd" field.
- <Rm> Is the general-purpose source register, encoded in the "Rm" field.
- <amount> Is the rotate amount, encoded in the "rotate" field. It can have the following values:
  - 0 when rotate = 00
  - 8 when rotate = 01
  - 16 when rotate = 10
  - 24 when rotate = 11
 ROR #<amount> can be omitted, meaning a rotate amount of 0. This is the preferred disassembly for rotate == 0b00.

### Operation for all encodings

```

1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 rotated = ROR(R[m], rotation);
4 bits(32) result;
5 result[15:0] = SignExtend(rotated[7:0], 16);
6 result[31:16] = SignExtend(rotated[23:16], 16);
7 R[d] = result;

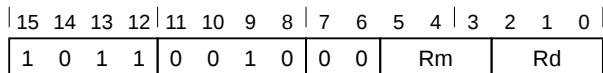
```

### C2.4.210 SXTB

Signed Extend Halfword. Signed Extend Halfword extracts a 16-bit value from a register, sign extends it to 32 bits, and writes the result to the destination register. The instruction can specify a rotation by 0, 8, 16, or 24 bits before extracting the 16-bit value.

#### T1

Armv8-M



#### T1 variant

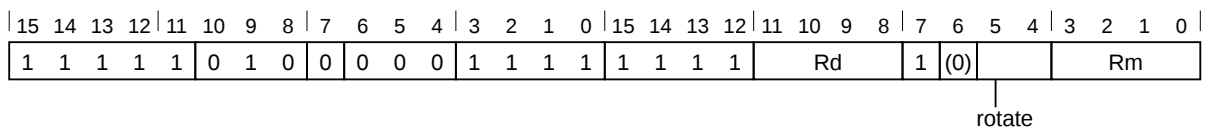
SXTB{<c>}{<q>} {<Rd>}, {<Rm>}

#### Decode for this encoding

```
1 d = UInt(Rd); m = UInt(Rm); rotation = 0;
```

#### T2

Armv8-M Main Extension only



#### T2 variant

SXTB{<c>}.W {<Rd>}, {<Rm>}  
 // <Rd>, <Rm> can be represented in T1  
 SXTB{<c>}{<q>} {<Rd>}, {<Rm>} {, ROR #<amount>}

#### Decode for this encoding

```
1 if !HaveMainExt() then UNDEFINED;
2 d = UInt(Rd); m = UInt(Rm); rotation = UInt(rotate:'000');
3 if d IN {13,15} || m IN {13,15} then UNPREDICTABLE;
```

#### Assembler symbols for all encodings

- <c> See [C1.2.5 Standard assembler syntax fields](#) on page 309.
- <q> See [C1.2.5 Standard assembler syntax fields](#) on page 309.
- <Rd> Is the general-purpose destination register, encoded in the "Rd" field.
- <Rm> Is the general-purpose source register, encoded in the "Rm" field.
- <amount> Is the rotate amount, encoded in the "rotate" field. It can have the following values:
  - 0 when rotate = 00
  - 8 when rotate = 01
  - 16 when rotate = 10
  - 24 when rotate = 11
 ROR #<amount> can be omitted, meaning a rotate amount of 0. This is the preferred disassembly for rotate == 0b00.

### Operation for all encodings

```
1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 rotated = ROR(R[m], rotation);
4 R[d] = SignExtend(rotated[15:0], 32);
```

### C2.4.211 TBB, TBH

Table Branch Byte or Halfword. Table Branch Byte causes a PC-relative forward branch using a table of single byte offsets. A base register provides a pointer to the table, and a second register supplies an index into the table. The branch length is twice the value of the byte returned from the table.

Table Branch Halfword causes a PC-relative forward branch using a table of single halfword offsets. A base register provides a pointer to the table, and a second register supplies an index into the table. The branch length is twice the value of the halfword returned from the table.

#### T1

Armv8-M Main Extension only

|    |    |    |    |    |    |   |   |   |   |   |   |    |   |   |   |     |     |     |     |     |     |     |     |   |   |   |   |    |   |   |   |
|----|----|----|----|----|----|---|---|---|---|---|---|----|---|---|---|-----|-----|-----|-----|-----|-----|-----|-----|---|---|---|---|----|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3  | 2 | 1 | 0 | 15  | 14  | 13  | 12  | 11  | 10  | 9   | 8   | 7 | 6 | 5 | 4 | 3  | 2 | 1 | 0 |
| 1  | 1  | 1  | 0  | 1  | 0  | 0 | 0 | 1 | 1 | 0 | 1 | Rn |   |   |   | (1) | (1) | (1) | (1) | (0) | (0) | (0) | (0) | 0 | 0 | 0 | H | Rm |   |   |   |

#### Byte variant

Applies when **H** == 0.

```
TBB{<c>}{<q>} [<Rn>, <Rm>]
// Outside or last in IT block
```

#### Halfword variant

Applies when **H** == 1.

```
TBH{<c>}{<q>} [<Rn>, <Rm>, LSL #1]
// Outside or last in IT block
```

#### Decode for this encoding

```
1 if !HaveMainExt() then UNDEFINED;
2 n = UInt(Rn); m = UInt(Rm); is_tbh = (H == '1');
3 if n == 13 || m IN {13,15} then UNPREDICTABLE;
4 if InITBlock() && !LastInITBlock() then UNPREDICTABLE;
```

#### Assembler symbols for all encodings

<c> See [C1.2.5 Standard assembler syntax fields](#) on page 309.  
 <q> See [C1.2.5 Standard assembler syntax fields](#) on page 309.  
 <Rn> Is the general-purpose base register holding the address of the table of branch lengths, encoded in the "Rn" field. The PC can be used. If it is, the table immediately follows this instruction.  
 <Rm> For the byte variant: is the general-purpose index register, encoded in the "Rm" field. This register contains an integer pointing to a single byte in the table. The offset in the table is the value of the index.  
 For the halfword variant: is the general-purpose index register, encoded in the "Rm" field. This register contains an integer pointing to a halfword in the table. The offset in the table is twice the value of the index.

#### Operation for all encodings

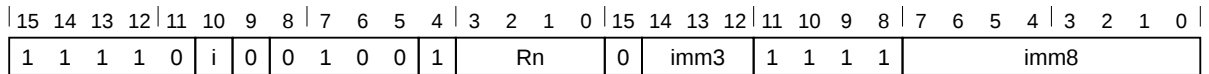
```
1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 if is_tbh then
4 halfwords = UInt(MemU[R[n]+LSL(R[m],1), 2]);
5 else
6 halfwords = UInt(MemU[R[n]+R[m], 1]);
7 BranchWritePC(PC + 2*halfwords);
```

### C2.4.212 TEQ (immediate)

Test Equivalence (immediate). Test Equivalence (immediate) performs an exclusive OR operation on a register value and an immediate value. It updates the condition flags based on the result, and discards the result.

#### T1

Armv8-M Main Extension only



#### T1 variant

TEQ{<c>}{<q>} <Rn>, #<const>

#### Decode for this encoding

```

1 if !HaveMainExt() then UNDEFINED;
2 n = UInt(Rn);
3 (imm32, carry) = T32ExpandImm_C(i:imm3:imm8, APSR.C);
4 if n IN {13,15} then UNPREDICTABLE;
```

#### Assembler symbols for all encodings

<c> See [C1.2.5 Standard assembler syntax fields](#) on page 309.  
 <q> See [C1.2.5 Standard assembler syntax fields](#) on page 309.  
 <Rn> Is the general-purpose source register, encoded in the "Rn" field.  
 <const> Is an immediate value derived from the 12-bit immediate that is encoded in the 'i:imm3:imm8' field. See [C1.5 Modified immediate constants](#) on page 324 for the range of values.

#### Operation for all encodings

```

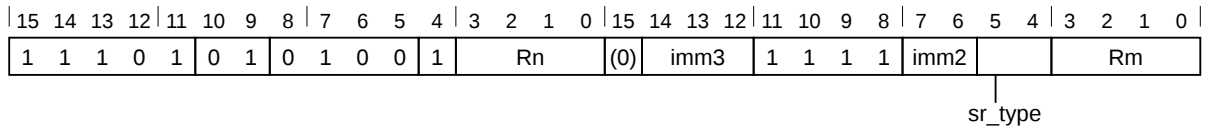
1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 result = R[n] EOR imm32;
4 APSR.N = result[31];
5 APSR.Z = IsZeroBit(result);
6 APSR.C = carry;
7 // APSR.V unchanged
```

### C2.4.213 TEQ (register)

Test Equivalence (register). Test Equivalence (register) performs an exclusive OR operation on a register value and an optionally-shifted register value. It updates the condition flags based on the result, and discards the result.

#### T1

Armv8-M Main Extension only



#### Rotate right with extend variant

Applies when `imm3 == 000 && imm2 == 00 && sr_type == 11`.

TEQ{<c>}{<q>} <Rn>, <Rm>, RRX

#### Shift or rotate by value variant

Applies when `!(imm3 == 000 && imm2 == 00 && sr_type == 11)`.

TEQ{<c>}{<q>} <Rn>, <Rm> {, <shift> #<amount>}

#### Decode for this encoding

```

1 if !HaveMainExt() then UNDEFINED;
2 n = UInt(Rn); m = UInt(Rm);
3 (shift_t, shift_n) = DecodeImmShift(sr_type, imm3:imm2);
4 if n IN {13,15} || m IN {13,15} then UNPREDICTABLE;

```

#### Assembler symbols for all encodings

- <c> See [C1.2.5 Standard assembler syntax fields](#) on page 309.
- <q> See [C1.2.5 Standard assembler syntax fields](#) on page 309.
- <Rn> Is the first general-purpose source register, encoded in the "Rn" field.
- <Rm> Is the second general-purpose source register, encoded in the "Rm" field.
- <shift> Is the type of shift to be applied to the second source register, encoded in the "sr\_type" field. It can have the following values:
  - LSL when sr\_type = 00
  - LSR when sr\_type = 01
  - ASR when sr\_type = 10
  - ROR when sr\_type = 11
- <amount> Is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR) encoded in the "imm3:imm2" field as <amount> modulo 32.

#### Operation for all encodings

```

1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 (shifted, carry) = Shift_C(R[m], shift_t, shift_n, APSR.C);
4 result = R[n] EOR shifted;
5 APSR.N = result[31];
6 APSR.Z = IsZeroBit(result);
7 APSR.C = carry;
8 // APSR.V unchanged

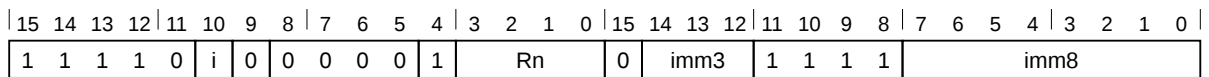
```

### C2.4.214 TST (immediate)

Test (immediate). Test (immediate) performs a bitwise AND operation on a register value and an immediate value. It updates the condition flags based on the result, and discards the result.

#### T1

Armv8-M Main Extension only



#### T1 variant

TST{<c>}{<q>} <Rn>, #<const>

#### Decode for this encoding

```

1 if !HaveMainExt() then UNDEFINED;
2 n = UInt(Rn);
3 (imm32, carry) = T32ExpandImm_C(i:imm3:imm8, APSR.C);
4 if n IN {13,15} then UNPREDICTABLE;

```

#### Assembler symbols for all encodings

<c> See [C1.2.5 Standard assembler syntax fields](#) on page 309.  
 <q> See [C1.2.5 Standard assembler syntax fields](#) on page 309.  
 <Rn> Is the general-purpose source register, encoded in the "Rn" field.  
 <const> Is an immediate value derived from the 12-bit immediate that is encoded in the 'i:imm3:imm8' field. See [C1.5 Modified immediate constants](#) on page 324 for the range of values.

#### Operation for all encodings

```

1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 result = R[n] AND imm32;
4 APSR.N = result[31];
5 APSR.Z = IsZeroBit(result);
6 APSR.C = carry;
7 // APSR.V unchanged

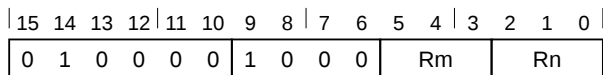
```

### C2.4.215 TST (register)

Test (register). Test (register) performs a bitwise AND operation on a register value and an optionally-shifted register value. It updates the condition flags based on the result, and discards the result.

#### T1

Armv8-M



#### T1 variant

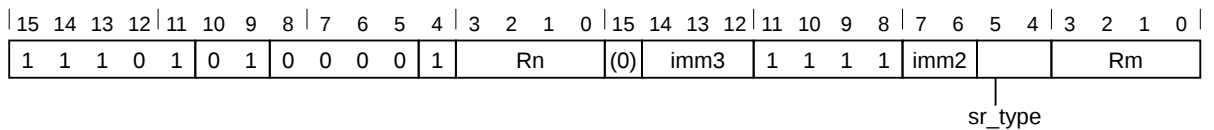
TST{<c>}{<q>} <Rn>, <Rm>

#### Decode for this encoding

```
1 n = UInt(Rn); m = UInt(Rm);
2 (shift_t, shift_n) = (SRType_LSL, 0);
```

#### T2

Armv8-M Main Extension only



#### Rotate right with extend variant

Applies when **imm3 == 000 && imm2 == 00 && sr\_type == 11**.

TST{<c>}{<q>} <Rn>, <Rm>, RRX

#### Shift or rotate by value variant

Applies when **!(imm3 == 000 && imm2 == 00 && sr\_type == 11)**.

TST{<c>}.W <Rn>, <Rm>  
 // <Rn>, <Rm> can be represented in T1  
 TST{<c>}{<q>} <Rn>, <Rm> {, <shift> #<amount>}

#### Decode for this encoding

```
1 if !HaveMainExt() then UNDEFINED;
2 n = UInt(Rn); m = UInt(Rm);
3 (shift_t, shift_n) = DecodeImmShift(sr_type, imm3:imm2);
4 if n IN {13,15} || m IN {13,15} then UNPREDICTABLE;
```

#### Assembler symbols for all encodings

- <c> See [C1.2.5 Standard assembler syntax fields](#) on page 309.
- <q> See [C1.2.5 Standard assembler syntax fields](#) on page 309.
- <Rn> Is the first general-purpose source register, encoded in the "Rn" field.
- <Rm> Is the second general-purpose source register, encoded in the "Rm" field.
- <shift> Is the type of shift to be applied to the second source register, encoded in the "sr\_type" field. It can have the following values:



LSL   when `sr_type = 00`  
 LSR   when `sr_type = 01`  
 ASR   when `sr_type = 10`  
 ROR   when `sr_type = 11`  
 <amount>   Is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR) encoded in the "imm3:imm2" field as <amount> modulo 32.

### Operation for all encodings

```

1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 (shifted, carry) = Shift_C(R[m], shift_t, shift_n, APSR.C);
4 result = R[n] AND shifted;
5 APSR.N = result[31];
6 APSR.Z = IsZeroBit(result);
7 APSR.C = carry;
8 // APSR.V unchanged

```

### C2.4.216 TT, TTT, TTA, TTAT

Test Target (Alternate Domain, Unprivileged). Test Target (TT) queries the Security state and access permissions of a memory location.

Test Target Unprivileged (TTT) queries the Security state and access permissions of a memory location for an unprivileged access to that location.

Test Target Alternate Domain (TTA) and Test Target Alternate Domain Unprivileged (TTAT) query the Security state and access permissions of a memory location for a Non-secure access to that location. These instructions are only valid when executing in Secure state, and are UNDEFINED if used from Non-secure state.

These instructions return the Security state and access permissions in the destination register. See TT\_RESP for the format of the destination register.

#### T1

Armv8-M

|    |    |    |    |    |    |   |   |   |   |   |   |    |   |   |   |    |    |    |    |     |     |     |     |     |     |   |   |   |   |   |   |
|----|----|----|----|----|----|---|---|---|---|---|---|----|---|---|---|----|----|----|----|-----|-----|-----|-----|-----|-----|---|---|---|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3  | 2 | 1 | 0 | 15 | 14 | 13 | 12 | 11  | 10  | 9   | 8   | 7   | 6   | 5 | 4 | 3 | 2 | 1 | 0 |
| 1  | 1  | 1  | 0  | 1  | 0  | 0 | 0 | 0 | 1 | 0 | 0 | Rn | 1 | 1 | 1 | 1  | Rd | A  | T  | (0) | (0) | (0) | (0) | (0) | (0) |   |   |   |   |   |   |

#### TT variant

Applies when **A == 0** && **T == 0**.

TT{<c>}{<q>} <Rd>, <Rn>

#### TTA variant

Applies when **A == 1** && **T == 0**.

TTA{<c>}{<q>} <Rd>, <Rn>

#### TTAT variant

Applies when **A == 1** && **T == 1**.

TTAT{<c>}{<q>} <Rd>, <Rn>

#### TTT variant

Applies when **A == 0** && **T == 1**.

TTT{<c>}{<q>} <Rd>, <Rn>

#### Decode for this encoding

```

1 d = UInt(Rd); n = UInt(Rn); alt = (A == '1'); forceunpriv = (T == '1');
2 if d IN {13,15} || n == 15 then UNPREDICTABLE;
3 if alt && !IsSecure() then UNDEFINED;

```

#### Assembler symbols for all encodings

- <c> See [C1.2.5 Standard assembler syntax fields](#) on page 309.
- <q> See [C1.2.5 Standard assembler syntax fields](#) on page 309.
- <Rd> Is the destination general-purpose register into which the status result of the target test is written, encoded in the "Rd" field.
- <Rn> Is the general-purpose base register, encoded in the "Rn" field.

### Operation for all encodings

```
1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 addr = R[n];
4 R[d] = TTResp(addr, alt, forceunpriv);
```

### C2.4.217 UADD16

Unsigned Add 16. Unsigned Add 16 performs two 16-bit unsigned integer additions, and writes the results to the destination register. It sets the APSR.GE bits according to the results of the additions.

#### T1

*Armv8-M DSP Extension only*

|    |    |    |    |    |    |   |   |   |   |   |   |    |   |   |   |    |    |    |    |    |    |   |   |   |   |   |   |    |   |   |   |
|----|----|----|----|----|----|---|---|---|---|---|---|----|---|---|---|----|----|----|----|----|----|---|---|---|---|---|---|----|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3  | 2 | 1 | 0 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3  | 2 | 1 | 0 |
| 1  | 1  | 1  | 1  | 1  | 0  | 1 | 0 | 1 | 0 | 0 | 1 | Rn |   |   |   | 1  | 1  | 1  | 1  | Rd |    |   |   | 0 | 1 | 0 | 0 | Rm |   |   |   |

#### T1 variant

UADD16{<c>}{<q>} {<Rd>}, {<Rn>}, <Rm>

#### Decode for this encoding

```

1 if !HaveDSPExt() then UNDEFINED;
2 d = UInt(Rd); n = UInt(Rn); m = UInt(Rm);
3 if d IN {13,15} || n IN {13,15} || m IN {13,15} then UNPREDICTABLE;

```

#### Assembler symbols for all encodings

<c> See [C1.2.5 Standard assembler syntax fields](#) on page 309.  
 <q> See [C1.2.5 Standard assembler syntax fields](#) on page 309.  
 <Rd> Is the general-purpose destination register, encoded in the "Rd" field.  
 <Rn> Is the first general-purpose source register, encoded in the "Rn" field.  
 <Rm> Is the second general-purpose source register, encoded in the "Rm" field.

#### Operation for all encodings

```

1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 sum1 = UInt(R[n][15:0]) + UInt(R[m][15:0]);
4 sum2 = UInt(R[n][31:16]) + UInt(R[m][31:16]);
5 R[d] = sum2[15:0] : sum1[15:0];
6 APSR.GE[1:0] = if sum1 >= 0x10000 then '11' else '00';
7 APSR.GE[3:2] = if sum2 >= 0x10000 then '11' else '00';

```

## C2.4.218 UADD8

Unsigned Add 8. Unsigned Add 8 performs four unsigned 8-bit integer additions, and writes the results to the destination register. It sets the APSR.GE bits according to the results of the additions.

### T1

*Armv8-M DSP Extension only*

|    |    |    |    |    |    |   |   |   |   |   |   |    |   |   |   |    |    |    |    |    |    |   |   |   |   |   |   |    |   |   |   |
|----|----|----|----|----|----|---|---|---|---|---|---|----|---|---|---|----|----|----|----|----|----|---|---|---|---|---|---|----|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3  | 2 | 1 | 0 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3  | 2 | 1 | 0 |
| 1  | 1  | 1  | 1  | 1  | 0  | 1 | 0 | 1 | 0 | 0 | 0 | Rn |   |   |   | 1  | 1  | 1  | 1  | Rd |    |   |   | 0 | 1 | 0 | 0 | Rm |   |   |   |

### T1 variant

UADD8 {<c>} {<q>} {<Rd>}, {<Rn>}, {<Rm>}

### Decode for this encoding

```

1 if !HaveDSPExt() then UNDEFINED;
2 d = UInt(Rd); n = UInt(Rn); m = UInt(Rm);
3 if d IN {13,15} || n IN {13,15} || m IN {13,15} then UNPREDICTABLE;

```

### Assembler symbols for all encodings

<c> See [C1.2.5 Standard assembler syntax fields](#) on page 309.  
 <q> See [C1.2.5 Standard assembler syntax fields](#) on page 309.  
 <Rd> Is the general-purpose destination register, encoded in the "Rd" field.  
 <Rn> Is the first general-purpose source register, encoded in the "Rn" field.  
 <Rm> Is the second general-purpose source register, encoded in the "Rm" field.

### Operation for all encodings

```

1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 sum1 = UInt(R[n][7:0]) + UInt(R[m][7:0]);
4 sum2 = UInt(R[n][15:8]) + UInt(R[m][15:8]);
5 sum3 = UInt(R[n][23:16]) + UInt(R[m][23:16]);
6 sum4 = UInt(R[n][31:24]) + UInt(R[m][31:24]);
7 R[d] = sum4[7:0] : sum3[7:0] : sum2[7:0] : sum1[7:0];
8 APSR.GE[0] = if sum1 >= 0x100 then '1' else '0';
9 APSR.GE[1] = if sum2 >= 0x100 then '1' else '0';
10 APSR.GE[2] = if sum3 >= 0x100 then '1' else '0';
11 APSR.GE[3] = if sum4 >= 0x100 then '1' else '0';

```

### C2.4.219 UASX

Unsigned Add and Subtract with Exchange. Unsigned Add and Subtract with Exchange exchanges the two halfwords of the second operand, performs one unsigned 16-bit integer addition and one unsigned 16-bit subtraction, and writes the results to the destination register. It sets the APSR.GE bits according to the results.

#### T1

Armv8-M DSP Extension only

|    |    |    |    |    |    |   |   |   |   |   |   |    |   |   |   |    |    |    |    |    |    |   |   |   |   |   |   |    |   |   |   |
|----|----|----|----|----|----|---|---|---|---|---|---|----|---|---|---|----|----|----|----|----|----|---|---|---|---|---|---|----|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3  | 2 | 1 | 0 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3  | 2 | 1 | 0 |
| 1  | 1  | 1  | 1  | 1  | 0  | 1 | 0 | 1 | 0 | 1 | 0 | Rn |   |   |   | 1  | 1  | 1  | 1  | Rd |    |   |   | 0 | 1 | 0 | 0 | Rm |   |   |   |

#### T1 variant

UASX{<c>}{<q>} {<Rd>}, {<Rn>}, <Rm>

#### Decode for this encoding

```

1 if !HaveDSPExt() then UNDEFINED;
2 d = UInt(Rd); n = UInt(Rn); m = UInt(Rm);
3 if d IN {13,15} || n IN {13,15} || m IN {13,15} then UNPREDICTABLE;

```

#### Assembler symbols for all encodings

<c> See [C1.2.5 Standard assembler syntax fields](#) on page 309.  
 <q> See [C1.2.5 Standard assembler syntax fields](#) on page 309.  
 <Rd> Is the general-purpose destination register, encoded in the "Rd" field.  
 <Rn> Is the first general-purpose source register, encoded in the "Rn" field.  
 <Rm> Is the second general-purpose source register, encoded in the "Rm" field.

#### Operation for all encodings

```

1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 diff = UInt(R[n][15:0]) - UInt(R[m][31:16]);
4 sum = UInt(R[n][31:16]) + UInt(R[m][15:0]);
5 R[d] = sum[15:0] : diff[15:0];
6 APSR.GE[1:0] = if diff >= 0 then '11' else '00';
7 APSR.GE[3:2] = if sum >= 0x10000 then '11' else '00';

```

## C2.4.220 UBFX

Unsigned Bit Field Extract. Unsigned Bit Field Extract extracts any number of adjacent bits at any position from one register, zero extends them to 32 bits, and writes the result to the destination register.

### T1

Armv8-M Main Extension only

|    |    |    |    |    |     |   |   |   |   |   |   |    |   |   |   |    |      |    |    |    |    |   |   |   |      |     |         |   |   |   |   |
|----|----|----|----|----|-----|---|---|---|---|---|---|----|---|---|---|----|------|----|----|----|----|---|---|---|------|-----|---------|---|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10  | 9 | 8 | 7 | 6 | 5 | 4 | 3  | 2 | 1 | 0 | 15 | 14   | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6    | 5   | 4       | 3 | 2 | 1 | 0 |
| 1  | 1  | 1  | 1  | 0  | (0) | 1 | 1 | 1 | 1 | 0 | 0 | Rn |   |   |   | 0  | imm3 |    |    |    | Rd |   |   |   | imm2 | (0) | widthm1 |   |   |   |   |

### T1 variant

UBFX{<c>}{<q>} <Rd>, <Rn>, #<lsb>, #<width>

### Decode for this encoding

```

1 if !HaveMainExt() then UNDEFINED;
2 d = UInt(Rd); n = UInt(Rn);
3 lsb = UInt(imm3:imm2); widthminus1 = UInt(widthm1);
4 msbit = lsb + widthminus1;
5 if msbit > 31 then UNPREDICTABLE;
6 if d IN {13,15} || n IN {13,15} then UNPREDICTABLE;

```

### CONSTRAINED UNPREDICTABLE behavior

If `msbit > 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The value in the destination register is UNKNOWN.

### Assembler symbols for all encodings

|         |                                                                                                                      |
|---------|----------------------------------------------------------------------------------------------------------------------|
| <c>     | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                                             |
| <q>     | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                                             |
| <Rd>    | Is the general-purpose destination register, encoded in the "Rd" field.                                              |
| <Rn>    | Is the general-purpose source register, encoded in the "Rn" field.                                                   |
| <lsb>   | Is the bit number of the least significant bit in the field, in the range 0 to 31, encoded in the "imm3:imm2" field. |
| <width> | Is the width of the field, in the range 1 to 32-<lsb>, encoded in the "widthm1" field as <width>-1.                  |

### Operation for all encodings

```

1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 if msbit <= 31 then
4 R[d] = ZeroExtend(R[n][msbit:lsb], 32);
5 else
6 R[d] = bits(32) UNKNOWN;

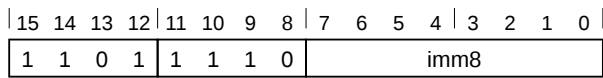
```

### C2.4.221 UDF

Permanently Undefined. Permanently Undefined generates an Undefined Instruction exception.

#### T1

Armv8-M



#### T1 variant

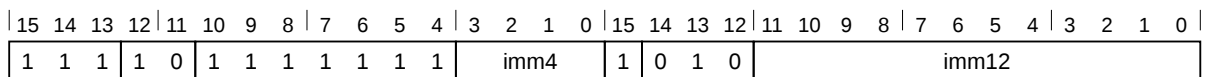
UDF{<c>}{<q>} {#}<imm>

#### Decode for this encoding

```
1 imm32 = ZeroExtend(imm8, 32);
2 // imm32 is for assembly and disassembly only, and is ignored by hardware.
```

#### T2

Armv8-M



#### T2 variant

UDF{<c>}.W {#}<imm>  
 // <imm> can be represented in T1  
 UDF{<c>}{<q>} {#}<imm>

#### Decode for this encoding

```
1 imm32 = ZeroExtend(imm4:imm12, 32);
2 // imm32 is for assembly and disassembly only, and is ignored by hardware.
```

#### Assembler symbols for all encodings

- <c> See [C1.2.5 Standard assembler syntax fields](#) on page 309. Arm deprecates using any <c> value other than AL.
- <q> See [C1.2.5 Standard assembler syntax fields](#) on page 309.
- <imm> For encoding T1: is an 8-bit unsigned immediate, in the range 0 to 255, encoded in the "imm8" field. The PE ignores the value of this constant.  
 For encoding T2: is a 16-bit unsigned immediate, in the range 0 to 65535, encoded in the "imm4:imm12" field. The PE ignores the value of this constant.

#### Operation for all encodings

```
1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 UNDEFINED;
```



## C2.4.222 UDIV

Unsigned Divide. Unsigned Divide divides a 32-bit unsigned integer register value by a 32-bit unsigned integer register value, and writes the result to the destination register. The condition flags are not affected.

### T1

Armv8-M

|    |    |    |    |    |    |   |   |   |   |   |   |    |   |   |   |     |     |     |     |    |    |   |   |   |   |   |   |    |   |   |   |
|----|----|----|----|----|----|---|---|---|---|---|---|----|---|---|---|-----|-----|-----|-----|----|----|---|---|---|---|---|---|----|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3  | 2 | 1 | 0 | 15  | 14  | 13  | 12  | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3  | 2 | 1 | 0 |
| 1  | 1  | 1  | 1  | 1  | 0  | 1 | 1 | 1 | 0 | 1 | 1 | Rn |   |   |   | (1) | (1) | (1) | (1) | Rd |    |   |   | 1 | 1 | 1 | 1 | Rm |   |   |   |

### T1 variant

UDIV{<c>}{<q>} {<Rd>}, {<Rn>}, <Rm>

### Decode for this encoding

```
1 d = UInt(Rd); n = UInt(Rn); m = UInt(Rm);
2 if d IN {13,15} || n IN {13,15} || m IN {13,15} then UNPREDICTABLE;
```

### Assembler symbols for all encodings

<c> See [C1.2.5 Standard assembler syntax fields](#) on page 309.  
 <q> See [C1.2.5 Standard assembler syntax fields](#) on page 309.  
 <Rd> Is the general-purpose destination register, encoded in the "Rd" field.  
 <Rn> Is the first general-purpose source register holding the dividend, encoded in the "Rn" field.  
 <Rm> Is the second general-purpose source register holding the divisor, encoded in the "Rm" field.

### Operation for all encodings

```
1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 if UInt(R[m]) == 0 then
4 if IntegerZeroDivideTrappingEnabled() then
5 GenerateIntegerZeroDivide();
6 else
7 result = 0;
8 else
9 result = RoundTowardsZero(Real(UInt(R[n])) / Real(UInt(R[m])));
10 R[d] = result[31:0];
```

### C2.4.223 UHADD16

Unsigned Halving Add 16. Unsigned Halving Add 16 performs two unsigned 16-bit integer additions, halves the results, and writes the results to the destination register.

#### T1

Armv8-M DSP Extension only

|    |    |    |    |    |    |   |   |   |   |   |   |    |   |   |   |    |    |    |    |    |    |   |   |   |   |   |   |    |   |   |   |
|----|----|----|----|----|----|---|---|---|---|---|---|----|---|---|---|----|----|----|----|----|----|---|---|---|---|---|---|----|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3  | 2 | 1 | 0 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3  | 2 | 1 | 0 |
| 1  | 1  | 1  | 1  | 1  | 0  | 1 | 0 | 1 | 0 | 0 | 1 | Rn |   |   |   | 1  | 1  | 1  | 1  | Rd |    |   |   | 0 | 1 | 1 | 0 | Rm |   |   |   |

#### T1 variant

UHADD16 {<c>} {<q>} {<Rd>}, {<Rn>}, {<Rm>}

#### Decode for this encoding

```

1 if !HaveDSPExt() then UNDEFINED;
2 d = UInt(Rd); n = UInt(Rn); m = UInt(Rm);
3 if d IN {13,15} || n IN {13,15} || m IN {13,15} then UNPREDICTABLE;

```

#### Assembler symbols for all encodings

<c> See [C1.2.5 Standard assembler syntax fields](#) on page 309.  
 <q> See [C1.2.5 Standard assembler syntax fields](#) on page 309.  
 <Rd> Is the general-purpose destination register, encoded in the "Rd" field.  
 <Rn> Is the first general-purpose source register, encoded in the "Rn" field.  
 <Rm> Is the second general-purpose source register, encoded in the "Rm" field.

#### Operation for all encodings

```

1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 sum1 = UInt(R[n][15:0]) + UInt(R[m][15:0]);
4 sum2 = UInt(R[n][31:16]) + UInt(R[m][31:16]);
5 R[d] = sum2[16:1] : sum1[16:1];

```

## C2.4.224 UHADD8

Unsigned Halving Add 8. Unsigned Halving Add 8 performs four unsigned 8-bit integer additions, halves the results, and writes the results to the destination register.

### T1

*Armv8-M DSP Extension only*

|    |    |    |    |    |    |   |   |   |   |   |   |    |   |   |   |    |    |    |    |    |    |   |   |   |   |   |   |    |   |   |   |
|----|----|----|----|----|----|---|---|---|---|---|---|----|---|---|---|----|----|----|----|----|----|---|---|---|---|---|---|----|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3  | 2 | 1 | 0 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3  | 2 | 1 | 0 |
| 1  | 1  | 1  | 1  | 1  | 0  | 1 | 0 | 1 | 0 | 0 | 0 | Rn |   |   |   | 1  | 1  | 1  | 1  | Rd |    |   |   | 0 | 1 | 1 | 0 | Rm |   |   |   |

### T1 variant

UHADD8 {<c>} {<q>} {<Rd>}, {<Rn>}, <Rm>

### Decode for this encoding

```

1 if !HaveDSPExt() then UNDEFINED;
2 d = UInt(Rd); n = UInt(Rn); m = UInt(Rm);
3 if d IN {13,15} || n IN {13,15} || m IN {13,15} then UNPREDICTABLE;

```

### Assembler symbols for all encodings

<c> See [C1.2.5 Standard assembler syntax fields](#) on page 309.  
 <q> See [C1.2.5 Standard assembler syntax fields](#) on page 309.  
 <Rd> Is the general-purpose destination register, encoded in the "Rd" field.  
 <Rn> Is the first general-purpose source register, encoded in the "Rn" field.  
 <Rm> Is the second general-purpose source register, encoded in the "Rm" field.

### Operation for all encodings

```

1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 sum1 = UInt(R[n][7:0]) + UInt(R[m][7:0]);
4 sum2 = UInt(R[n][15:8]) + UInt(R[m][15:8]);
5 sum3 = UInt(R[n][23:16]) + UInt(R[m][23:16]);
6 sum4 = UInt(R[n][31:24]) + UInt(R[m][31:24]);
7 R[d] = sum4[8:1] : sum3[8:1] : sum2[8:1] : sum1[8:1];

```

## C2.4.225 UHASX

Unsigned Halving Add and Subtract with Exchange. Unsigned Halving Add and Subtract with Exchange exchanges the two halfwords of the second operand, performs one unsigned 16-bit integer addition and one unsigned 16-bit subtraction, halves the results, and writes the results to the destination register.

### T1

Armv8-M DSP Extension only

|    |    |    |    |    |    |   |   |   |   |   |   |    |   |   |   |    |    |    |    |    |    |   |   |   |   |   |   |    |   |   |   |
|----|----|----|----|----|----|---|---|---|---|---|---|----|---|---|---|----|----|----|----|----|----|---|---|---|---|---|---|----|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3  | 2 | 1 | 0 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3  | 2 | 1 | 0 |
| 1  | 1  | 1  | 1  | 1  | 0  | 1 | 0 | 1 | 0 | 1 | 0 | Rn |   |   |   | 1  | 1  | 1  | 1  | Rd |    |   |   | 0 | 1 | 1 | 0 | Rm |   |   |   |

### T1 variant

UHASX{<c>}{<q>} {<Rd>}, {<Rn>}, {<Rm>}

### Decode for this encoding

```

1 if !HaveDSPExt() then UNDEFINED;
2 d = UInt(Rd); n = UInt(Rn); m = UInt(Rm);
3 if d IN {13,15} || n IN {13,15} || m IN {13,15} then UNPREDICTABLE;

```

### Assembler symbols for all encodings

<c> See [C1.2.5 Standard assembler syntax fields](#) on page 309.  
 <q> See [C1.2.5 Standard assembler syntax fields](#) on page 309.  
 <Rd> Is the general-purpose destination register, encoded in the "Rd" field.  
 <Rn> Is the first general-purpose source register, encoded in the "Rn" field.  
 <Rm> Is the second general-purpose source register, encoded in the "Rm" field.

### Operation for all encodings

```

1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 diff = UInt(R[n][15:0]) - UInt(R[m][31:16]);
4 sum = UInt(R[n][31:16]) + UInt(R[m][15:0]);
5 R[d] = sum[16:1] : diff[16:1];

```

## C2.4.226 UHSAX

Unsigned Halving Subtract and Add with Exchange. Unsigned Halving Subtract and Add with Exchange exchanges the two halfwords of the second operand, performs one unsigned 16-bit integer subtraction and one unsigned 16-bit addition, halves the results, and writes the results to the destination register.

### T1

Armv8-M DSP Extension only

|    |    |    |    |    |    |   |   |   |   |   |   |    |   |   |   |    |    |    |    |    |    |   |   |   |   |   |   |    |   |   |   |
|----|----|----|----|----|----|---|---|---|---|---|---|----|---|---|---|----|----|----|----|----|----|---|---|---|---|---|---|----|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3  | 2 | 1 | 0 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3  | 2 | 1 | 0 |
| 1  | 1  | 1  | 1  | 1  | 0  | 1 | 0 | 1 | 1 | 1 | 0 | Rn |   |   |   | 1  | 1  | 1  | 1  | Rd |    |   |   | 0 | 1 | 1 | 0 | Rm |   |   |   |

### T1 variant

UHSAX{<c>}{<q>} {<Rd>}, {<Rn>}, {<Rm>}

### Decode for this encoding

```

1 if !HaveDSPExt() then UNDEFINED;
2 d = UInt(Rd); n = UInt(Rn); m = UInt(Rm);
3 if d IN {13,15} || n IN {13,15} || m IN {13,15} then UNPREDICTABLE;

```

### Assembler symbols for all encodings

<c> See [C1.2.5 Standard assembler syntax fields](#) on page 309.  
 <q> See [C1.2.5 Standard assembler syntax fields](#) on page 309.  
 <Rd> Is the general-purpose destination register, encoded in the "Rd" field.  
 <Rn> Is the first general-purpose source register, encoded in the "Rn" field.  
 <Rm> Is the second general-purpose source register, encoded in the "Rm" field.

### Operation for all encodings

```

1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 sum = UInt(R[n][15:0]) + UInt(R[m][31:16]);
4 diff = UInt(R[n][31:16]) - UInt(R[m][15:0]);
5 R[d] = diff[16:1] : sum[16:1];

```

## C2.4.227 UHSUB16

Unsigned Halving Subtract 16. Unsigned Halving Subtract 16 performs two unsigned 16-bit integer subtractions, halves the results, and writes the results to the destination register.

### T1

*Armv8-M DSP Extension only*

|    |    |    |    |    |    |   |   |   |   |   |   |    |   |   |   |    |    |    |    |    |    |   |   |   |   |   |   |    |   |   |   |
|----|----|----|----|----|----|---|---|---|---|---|---|----|---|---|---|----|----|----|----|----|----|---|---|---|---|---|---|----|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3  | 2 | 1 | 0 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3  | 2 | 1 | 0 |
| 1  | 1  | 1  | 1  | 1  | 0  | 1 | 0 | 1 | 1 | 0 | 1 | Rn |   |   |   | 1  | 1  | 1  | 1  | Rd |    |   |   | 0 | 1 | 1 | 0 | Rm |   |   |   |

### T1 variant

UHSUB16{<c>}{<q>} {<Rd>}, {<Rn>}, {<Rm>}

### Decode for this encoding

```

1 if !HaveDSPExt() then UNDEFINED;
2 d = UInt(Rd); n = UInt(Rn); m = UInt(Rm);
3 if d IN {13,15} || n IN {13,15} || m IN {13,15} then UNPREDICTABLE;

```

### Assembler symbols for all encodings

<c> See [C1.2.5 Standard assembler syntax fields](#) on page 309.  
 <q> See [C1.2.5 Standard assembler syntax fields](#) on page 309.  
 <Rd> Is the general-purpose destination register, encoded in the "Rd" field.  
 <Rn> Is the first general-purpose source register, encoded in the "Rn" field.  
 <Rm> Is the second general-purpose source register, encoded in the "Rm" field.

### Operation for all encodings

```

1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 diff1 = UInt(R[n][15:0]) - UInt(R[m][15:0]);
4 diff2 = UInt(R[n][31:16]) - UInt(R[m][31:16]);
5 R[d] = diff2[16:1] : diff1[16:1];

```

## C2.4.228 UHSUB8

Unsigned Halving Subtract 8. Unsigned Halving Subtract 8 performs four unsigned 8-bit integer subtractions, halves the results, and writes the results to the destination register.

### T1

*Armv8-M DSP Extension only*

|    |    |    |    |    |    |   |   |   |   |   |   |    |   |   |   |    |    |    |    |    |    |   |   |   |   |   |   |    |   |   |   |
|----|----|----|----|----|----|---|---|---|---|---|---|----|---|---|---|----|----|----|----|----|----|---|---|---|---|---|---|----|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3  | 2 | 1 | 0 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3  | 2 | 1 | 0 |
| 1  | 1  | 1  | 1  | 1  | 0  | 1 | 0 | 1 | 1 | 0 | 0 | Rn |   |   |   | 1  | 1  | 1  | 1  | Rd |    |   |   | 0 | 1 | 1 | 0 | Rm |   |   |   |

### T1 variant

UHSUB8{<c>}{<q>} {<Rd>}, {<Rn>}, <Rm>

### Decode for this encoding

```

1 if !HaveDSPExt() then UNDEFINED;
2 d = UInt(Rd); n = UInt(Rn); m = UInt(Rm);
3 if d IN {13,15} || n IN {13,15} || m IN {13,15} then UNPREDICTABLE;

```

### Assembler symbols for all encodings

<c> See [C1.2.5 Standard assembler syntax fields](#) on page 309.  
 <q> See [C1.2.5 Standard assembler syntax fields](#) on page 309.  
 <Rd> Is the general-purpose destination register, encoded in the "Rd" field.  
 <Rn> Is the first general-purpose source register, encoded in the "Rn" field.  
 <Rm> Is the second general-purpose source register, encoded in the "Rm" field.

### Operation for all encodings

```

1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 diff1 = UInt(R[n][7:0]) - UInt(R[m][7:0]);
4 diff2 = UInt(R[n][15:8]) - UInt(R[m][15:8]);
5 diff3 = UInt(R[n][23:16]) - UInt(R[m][23:16]);
6 diff4 = UInt(R[n][31:24]) - UInt(R[m][31:24]);
7 R[d] = diff4[8:1] : diff3[8:1] : diff2[8:1] : diff1[8:1];

```

## C2.4.229 UMAAL

Unsigned Multiply Accumulate Accumulate Long. Unsigned Multiply Accumulate Accumulate Long multiplies two unsigned 32-bit values to produce a 64-bit value, adds two unsigned 32-bit values, and writes the 64-bit result to two registers.

### T1

Armv8-M DSP Extension only

|    |    |    |    |    |    |   |   |   |   |   |   |    |   |   |   |      |    |    |    |      |    |   |   |         |   |   |   |    |   |   |   |
|----|----|----|----|----|----|---|---|---|---|---|---|----|---|---|---|------|----|----|----|------|----|---|---|---------|---|---|---|----|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3  | 2 | 1 | 0 | 15   | 14 | 13 | 12 | 11   | 10 | 9 | 8 | 7       | 6 | 5 | 4 | 3  | 2 | 1 | 0 |
| 1  | 1  | 1  | 1  | 1  | 0  | 1 | 1 | 1 | 1 | 1 | 0 | Rn |   |   |   | RdLo |    |    |    | RdHi |    |   |   | 0 1 1 0 |   |   |   | Rm |   |   |   |

### T1 variant

UMAAL{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>

### Decode for this encoding

```

1 if !HaveDSPExt() then UNDEFINED;
2 dLo = UInt(RdLo); dHi = UInt(RdHi); n = UInt(Rn); m = UInt(Rm);
3 if dLo IN {13,15} || dHi IN {13,15} || n IN {13,15} || m IN {13,15} then UNPREDICTABLE;
4 if dHi == dLo then UNPREDICTABLE;

```

### CONSTRAINED UNPREDICTABLE behavior

If `dHi == dLo`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The value in the destination register is UNKNOWN.

### Assembler symbols for all encodings

|        |                                                                                                                                                                 |
|--------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <c>    | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                                                                                        |
| <q>    | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                                                                                        |
| <RdLo> | Is the general-purpose source register holding the first addend and the destination register for the lower 32 bits of the result, encoded in the "RdLo" field.  |
| <RdHi> | Is the general-purpose source register holding the second addend and the destination register for the upper 32 bits of the result, encoded in the "RdHi" field. |
| <Rn>   | Is the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.                                                               |
| <Rm>   | Is the second general-purpose source register holding the multiplier, encoded in the "Rm" field.                                                                |

### Operation for all encodings

```

1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 result = UInt(R[n]) * UInt(R[m]) + UInt(R[dHi]) + UInt(R[dLo]);
4 R[dHi] = result[63:32];
5 R[dLo] = result[31:0];

```



## C2.4.230 UMLAL

Unsigned Multiply Accumulate Long. Unsigned Multiply Accumulate Long multiplies two unsigned 32-bit values to produce a 64-bit value, and accumulates this with a 64-bit value.

### T1

Armv8-M Main Extension only

|    |    |    |    |    |    |   |   |   |   |   |   |    |   |   |   |      |    |    |    |      |    |   |   |         |   |   |   |    |   |   |   |
|----|----|----|----|----|----|---|---|---|---|---|---|----|---|---|---|------|----|----|----|------|----|---|---|---------|---|---|---|----|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3  | 2 | 1 | 0 | 15   | 14 | 13 | 12 | 11   | 10 | 9 | 8 | 7       | 6 | 5 | 4 | 3  | 2 | 1 | 0 |
| 1  | 1  | 1  | 1  | 1  | 0  | 1 | 1 | 1 | 1 | 1 | 0 | Rn |   |   |   | RdLo |    |    |    | RdHi |    |   |   | 0 0 0 0 |   |   |   | Rm |   |   |   |

### T1 variant

UMLAL{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>

### Decode for this encoding

```

1 if !HaveMainExt() then UNDEFINED;
2 dLo = UInt(RdLo); dHi = UInt(RdHi); n = UInt(Rn); m = UInt(Rm); setflags = FALSE;
3 if dLo IN {13,15} || dHi IN {13,15} || n IN {13,15} || m IN {13,15} then UNPREDICTABLE;
4 if dHi == dLo then UNPREDICTABLE;

```

### CONSTRAINED UNPREDICTABLE behavior

If `dHi == dLo`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The value in the destination register is UNKNOWN.

### Assembler symbols for all encodings

|        |                                                                                                                                                                                |
|--------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <c>    | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                                                                                                       |
| <q>    | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                                                                                                       |
| <RdLo> | Is the general-purpose source register holding the lower 32 bits of the addend, and the destination register for the lower 32 bits of the result, encoded in the "RdLo" field. |
| <RdHi> | Is the general-purpose source register holding the upper 32 bits of the addend, and the destination register for the upper 32 bits of the result, encoded in the "RdHi" field. |
| <Rn>   | Is the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.                                                                              |
| <Rm>   | Is the second general-purpose source register holding the multiplier, encoded in the "Rm" field.                                                                               |

### Operation for all encodings

```

1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 result = UInt(R[n]) * UInt(R[m]) + UInt(R[dHi]:R[dLo]);
4 R[dHi] = result[63:32];
5 R[dLo] = result[31:0];

```

### C2.4.231 UMULL

Unsigned Multiply Long. Unsigned Multiply Long multiplies two 32-bit unsigned values to produce a 64-bit result.

#### T1

Armv8-M Main Extension only

|    |    |    |    |    |    |   |   |   |   |   |   |    |   |   |   |      |    |    |    |      |    |   |   |   |   |   |   |    |   |   |   |
|----|----|----|----|----|----|---|---|---|---|---|---|----|---|---|---|------|----|----|----|------|----|---|---|---|---|---|---|----|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3  | 2 | 1 | 0 | 15   | 14 | 13 | 12 | 11   | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3  | 2 | 1 | 0 |
| 1  | 1  | 1  | 1  | 1  | 0  | 1 | 1 | 1 | 0 | 1 | 0 | Rn |   |   |   | RdLo |    |    |    | RdHi |    |   |   | 0 | 0 | 0 | 0 | Rm |   |   |   |

#### T1 variant

UMULL{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>

#### Decode for this encoding

```

1 if !HaveMainExt() then UNDEFINED;
2 dLo = UInt(RdLo); dHi = UInt(RdHi); n = UInt(Rn); m = UInt(Rm); setflags = FALSE;
3 if dLo IN {13,15} || dHi IN {13,15} || n IN {13,15} || m IN {13,15} then UNPREDICTABLE;
4 if dHi == dLo then UNPREDICTABLE;

```

#### CONSTRAINED UNPREDICTABLE behavior

If `dHi == dLo`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The value in the destination register is UNKNOWN.

#### Assembler symbols for all encodings

|        |                                                                                                               |
|--------|---------------------------------------------------------------------------------------------------------------|
| <c>    | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                                      |
| <q>    | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                                      |
| <RdLo> | Is the general-purpose destination register for the lower 32 bits of the result, encoded in the "RdLo" field. |
| <RdHi> | Is the general-purpose destination register for the upper 32 bits of the result, encoded in the "RdHi" field. |
| <Rn>   | Is the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.             |
| <Rm>   | Is the second general-purpose source register holding the multiplier, encoded in the "Rm" field.              |

#### Operation for all encodings

```

1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 result = UInt(R[n]) * UInt(R[m]);
4 R[dHi] = result[63:32];
5 R[dLo] = result[31:0];

```

### C2.4.232 UQADD16

Unsigned Saturating Add 16. Unsigned Saturating Add 16 performs two unsigned 16-bit integer additions, saturates the results to the 16-bit unsigned integer range 0 to  $2^{16}-1$ , and writes the results to the destination register.

#### T1

Armv8-M DSP Extension only

|    |    |    |    |    |    |   |   |   |   |   |   |    |   |   |   |    |    |    |    |    |    |   |   |   |   |   |   |    |   |   |   |
|----|----|----|----|----|----|---|---|---|---|---|---|----|---|---|---|----|----|----|----|----|----|---|---|---|---|---|---|----|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3  | 2 | 1 | 0 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3  | 2 | 1 | 0 |
| 1  | 1  | 1  | 1  | 1  | 0  | 1 | 0 | 1 | 0 | 0 | 1 | Rn |   |   |   | 1  | 1  | 1  | 1  | Rd |    |   |   | 0 | 1 | 0 | 1 | Rm |   |   |   |

#### T1 variant

UQADD16 {<c>} {<q>} {<Rd>}, {<Rn>}, {<Rm>}

#### Decode for this encoding

```

1 if !HaveDSPExt() then UNDEFINED;
2 d = UInt(Rd); n = UInt(Rn); m = UInt(Rm);
3 if d IN {13,15} || n IN {13,15} || m IN {13,15} then UNPREDICTABLE;

```

#### Assembler symbols for all encodings

<c> See [C1.2.5 Standard assembler syntax fields](#) on page 309.  
 <q> See [C1.2.5 Standard assembler syntax fields](#) on page 309.  
 <Rd> Is the general-purpose destination register, encoded in the "Rd" field.  
 <Rn> Is the first general-purpose source register, encoded in the "Rn" field.  
 <Rm> Is the second general-purpose source register, encoded in the "Rm" field.

#### Operation for all encodings

```

1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 sum1 = UInt(R[n][15:0]) + UInt(R[m][15:0]);
4 sum2 = UInt(R[n][31:16]) + UInt(R[m][31:16]);
5 bits(32) result;
6 result[15:0] = UnsignedSat(sum1, 16);
7 result[31:16] = UnsignedSat(sum2, 16);
8 R[d] = result;

```

### C2.4.233 UQADD8

Unsigned Saturating Add 8. Unsigned Saturating Add 8 performs four unsigned 8-bit integer additions, saturates the results to the 8-bit unsigned integer range 0 to  $2^8-1$ , and writes the results to the destination register.

#### T1

Armv8-M DSP Extension only

|    |    |    |    |    |    |   |   |   |   |   |   |    |   |   |   |    |    |    |    |    |    |   |   |   |   |   |   |    |   |   |   |
|----|----|----|----|----|----|---|---|---|---|---|---|----|---|---|---|----|----|----|----|----|----|---|---|---|---|---|---|----|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3  | 2 | 1 | 0 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3  | 2 | 1 | 0 |
| 1  | 1  | 1  | 1  | 1  | 0  | 1 | 0 | 1 | 0 | 0 | 0 | Rn |   |   |   | 1  | 1  | 1  | 1  | Rd |    |   |   | 0 | 1 | 0 | 1 | Rm |   |   |   |

#### T1 variant

UQADD8 {<c>} {<q>} {<Rd>}, {<Rn>}, {<Rm>}

#### Decode for this encoding

```

1 if !HaveDSPExt() then UNDEFINED;
2 d = UInt(Rd); n = UInt(Rn); m = UInt(Rm);
3 if d IN {13,15} || n IN {13,15} || m IN {13,15} then UNPREDICTABLE;

```

#### Assembler symbols for all encodings

<c> See [C1.2.5 Standard assembler syntax fields](#) on page 309.  
 <q> See [C1.2.5 Standard assembler syntax fields](#) on page 309.  
 <Rd> Is the general-purpose destination register, encoded in the "Rd" field.  
 <Rn> Is the first general-purpose source register, encoded in the "Rn" field.  
 <Rm> Is the second general-purpose source register, encoded in the "Rm" field.

#### Operation for all encodings

```

1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 sum1 = UInt(R[n][7:0]) + UInt(R[m][7:0]);
4 sum2 = UInt(R[n][15:8]) + UInt(R[m][15:8]);
5 sum3 = UInt(R[n][23:16]) + UInt(R[m][23:16]);
6 sum4 = UInt(R[n][31:24]) + UInt(R[m][31:24]);
7 bits(32) result;
8 result[7:0] = UnsignedSat(sum1, 8);
9 result[15:8] = UnsignedSat(sum2, 8);
10 result[23:16] = UnsignedSat(sum3, 8);
11 result[31:24] = UnsignedSat(sum4, 8);
12 R[d] = result;

```

### C2.4.234 UQASX

Unsigned Saturating Add and Subtract with Exchange. Unsigned Saturating Add and Subtract with Exchange exchanges the two halfwords of the second operand, performs one unsigned 16-bit integer addition and one unsigned 16-bit subtraction, saturates the results to the 16-bit unsigned integer range 0 to  $2^{16}-1$ , and writes the results to the destination register.

#### T1

Armv8-M DSP Extension only

|    |    |    |    |    |    |   |   |   |   |   |   |    |   |   |   |    |    |    |    |    |    |   |   |   |   |   |   |    |   |   |   |
|----|----|----|----|----|----|---|---|---|---|---|---|----|---|---|---|----|----|----|----|----|----|---|---|---|---|---|---|----|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3  | 2 | 1 | 0 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3  | 2 | 1 | 0 |
| 1  | 1  | 1  | 1  | 1  | 0  | 1 | 0 | 1 | 0 | 1 | 0 | Rn |   |   |   | 1  | 1  | 1  | 1  | Rd |    |   |   | 0 | 1 | 0 | 1 | Rm |   |   |   |

#### T1 variant

UQASX{<c>}{<q>} {<Rd>}, {<Rn>}, {<Rm>}

#### Decode for this encoding

```
1 if !HaveDSPExt() then UNDEFINED;
2 d = UInt(Rd); n = UInt(Rn); m = UInt(Rm);
3 if d IN {13,15} || n IN {13,15} || m IN {13,15} then UNPREDICTABLE;
```

#### Assembler symbols for all encodings

<c> See [C1.2.5 Standard assembler syntax fields](#) on page 309.  
 <q> See [C1.2.5 Standard assembler syntax fields](#) on page 309.  
 <Rd> Is the general-purpose destination register, encoded in the "Rd" field.  
 <Rn> Is the first general-purpose source register, encoded in the "Rn" field.  
 <Rm> Is the second general-purpose source register, encoded in the "Rm" field.

#### Operation for all encodings

```
1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 diff = UInt(R[n][15:0]) - UInt(R[m][31:16]);
4 sum = UInt(R[n][31:16]) + UInt(R[m][15:0]);
5 bits(32) result;
6 result[15:0] = UnsignedSat(diff, 16);
7 result[31:16] = UnsignedSat(sum, 16);
8 R[d] = result;
```

### C2.4.235 UQSAX

Unsigned Saturating Subtract and Add with Exchange. Unsigned Saturating Subtract and Add with Exchange exchanges the two halfwords of the second operand, performs one unsigned 16-bit integer subtraction and one unsigned 16-bit addition, saturates the results to the 16-bit unsigned integer range 0 to  $2^{16}-1$ , and writes the results to the destination register.

#### T1

Armv8-M DSP Extension only

|    |    |    |    |    |    |   |   |   |   |   |   |    |   |   |   |    |    |    |    |    |    |   |   |   |   |   |   |    |   |   |   |
|----|----|----|----|----|----|---|---|---|---|---|---|----|---|---|---|----|----|----|----|----|----|---|---|---|---|---|---|----|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3  | 2 | 1 | 0 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3  | 2 | 1 | 0 |
| 1  | 1  | 1  | 1  | 1  | 0  | 1 | 0 | 1 | 1 | 1 | 0 | Rn |   |   |   | 1  | 1  | 1  | 1  | Rd |    |   |   | 0 | 1 | 0 | 1 | Rm |   |   |   |

#### T1 variant

UQSAX{<c>}{<q>} {<Rd>}, {<Rn>}, {<Rm>}

#### Decode for this encoding

```

1 if !HaveDSPExt() then UNDEFINED;
2 d = UInt(Rd); n = UInt(Rn); m = UInt(Rm);
3 if d IN {13,15} || n IN {13,15} || m IN {13,15} then UNPREDICTABLE;

```

#### Assembler symbols for all encodings

<c> See [C1.2.5 Standard assembler syntax fields](#) on page 309.  
 <q> See [C1.2.5 Standard assembler syntax fields](#) on page 309.  
 <Rd> Is the general-purpose destination register, encoded in the "Rd" field.  
 <Rn> Is the first general-purpose source register, encoded in the "Rn" field.  
 <Rm> Is the second general-purpose source register, encoded in the "Rm" field.

#### Operation for all encodings

```

1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 sum = UInt(R[n][15:0]) + UInt(R[m][31:16]);
4 diff = UInt(R[n][31:16]) - UInt(R[m][15:0]);
5 bits(32) result;
6 result[15:0] = UnsignedSat(sum, 16);
7 result[31:16] = UnsignedSat(diff, 16);
8 R[d] = result;

```

### C2.4.236 UQSUB16

Unsigned Saturating Subtract 16. Unsigned Saturating Subtract 16 performs two unsigned 16-bit integer subtractions, saturates the results to the 16-bit unsigned integer range 0 to  $2^{16}-1$ , and writes the results to the destination register.

#### T1

Armv8-M DSP Extension only

|    |    |    |    |    |    |   |   |   |   |   |   |    |   |   |   |    |    |    |    |    |    |   |   |   |   |   |   |    |   |   |   |
|----|----|----|----|----|----|---|---|---|---|---|---|----|---|---|---|----|----|----|----|----|----|---|---|---|---|---|---|----|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3  | 2 | 1 | 0 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3  | 2 | 1 | 0 |
| 1  | 1  | 1  | 1  | 1  | 0  | 1 | 0 | 1 | 1 | 0 | 1 | Rn |   |   |   | 1  | 1  | 1  | 1  | Rd |    |   |   | 0 | 1 | 0 | 1 | Rm |   |   |   |

#### T1 variant

UQSUB16 {<c>} {<q>} {<Rd>}, {<Rn>}, {<Rm>}

#### Decode for this encoding

```

1 if !HaveDSPExt() then UNDEFINED;
2 d = UInt(Rd); n = UInt(Rn); m = UInt(Rm);
3 if d IN {13,15} || n IN {13,15} || m IN {13,15} then UNPREDICTABLE;

```

#### Assembler symbols for all encodings

<c> See [C1.2.5 Standard assembler syntax fields](#) on page 309.  
 <q> See [C1.2.5 Standard assembler syntax fields](#) on page 309.  
 <Rd> Is the general-purpose destination register, encoded in the "Rd" field.  
 <Rn> Is the first general-purpose source register, encoded in the "Rn" field.  
 <Rm> Is the second general-purpose source register, encoded in the "Rm" field.

#### Operation for all encodings

```

1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 diff1 = UInt(R[n][15:0]) - UInt(R[m][15:0]);
4 diff2 = UInt(R[n][31:16]) - UInt(R[m][31:16]);
5 bits(32) result;
6 result[15:0] = UnsignedSat(diff1, 16);
7 result[31:16] = UnsignedSat(diff2, 16);
8 R[d] = result;

```

### C2.4.237 UQSUB8

Unsigned Saturating Subtract 8. Unsigned Saturating Subtract 8 performs four unsigned 8-bit integer subtractions, saturates the results to the 8-bit unsigned integer range 0 to  $2^8-1$ , and writes the results to the destination register.

#### T1

Armv8-M DSP Extension only

|    |    |    |    |    |    |   |   |   |   |   |   |    |   |   |   |    |    |    |    |    |    |   |   |   |   |   |   |    |   |   |   |
|----|----|----|----|----|----|---|---|---|---|---|---|----|---|---|---|----|----|----|----|----|----|---|---|---|---|---|---|----|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3  | 2 | 1 | 0 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3  | 2 | 1 | 0 |
| 1  | 1  | 1  | 1  | 1  | 0  | 1 | 0 | 1 | 1 | 0 | 0 | Rn |   |   |   | 1  | 1  | 1  | 1  | Rd |    |   |   | 0 | 1 | 0 | 1 | Rm |   |   |   |

#### T1 variant

UQSUB8 {<c>} {<q>} {<Rd>}, {<Rn>}, <Rm>

#### Decode for this encoding

```

1 if !HaveDSPExt() then UNDEFINED;
2 d = UInt(Rd); n = UInt(Rn); m = UInt(Rm);
3 if d IN {13,15} || n IN {13,15} || m IN {13,15} then UNPREDICTABLE;

```

#### Assembler symbols for all encodings

<c> See [C1.2.5 Standard assembler syntax fields](#) on page 309.  
 <q> See [C1.2.5 Standard assembler syntax fields](#) on page 309.  
 <Rd> Is the general-purpose destination register, encoded in the "Rd" field.  
 <Rn> Is the first general-purpose source register, encoded in the "Rn" field.  
 <Rm> Is the second general-purpose source register, encoded in the "Rm" field.

#### Operation for all encodings

```

1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 diff1 = UInt(R[n][7:0]) - UInt(R[m][7:0]);
4 diff2 = UInt(R[n][15:8]) - UInt(R[m][15:8]);
5 diff3 = UInt(R[n][23:16]) - UInt(R[m][23:16]);
6 diff4 = UInt(R[n][31:24]) - UInt(R[m][31:24]);
7 bits(32) result;
8 result[7:0] = UnsignedSat(diff1, 8);
9 result[15:8] = UnsignedSat(diff2, 8);
10 result[23:16] = UnsignedSat(diff3, 8);
11 result[31:24] = UnsignedSat(diff4, 8);
12 R[d] = result;

```



## C2.4.238 USAD8

Unsigned Sum of Absolute Differences. Unsigned Sum of Absolute Differences performs four unsigned 8-bit subtractions, and adds the absolute values of the differences together.

### T1

Armv8-M DSP Extension only

|    |    |    |    |    |    |   |   |   |   |   |   |    |   |   |   |    |    |    |    |    |    |   |   |   |   |   |   |    |   |   |   |
|----|----|----|----|----|----|---|---|---|---|---|---|----|---|---|---|----|----|----|----|----|----|---|---|---|---|---|---|----|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3  | 2 | 1 | 0 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3  | 2 | 1 | 0 |
| 1  | 1  | 1  | 1  | 1  | 0  | 1 | 1 | 0 | 1 | 1 | 1 | Rn |   |   |   | 1  | 1  | 1  | 1  | Rd |    |   |   | 0 | 0 | 0 | 0 | Rm |   |   |   |

### T1 variant

USAD8 {<c>} {<q>} {<Rd>}, {<Rn>}, {<Rm>}

### Decode for this encoding

```

1 if !HaveDSPExt() then UNDEFINED;
2 d = UInt(Rd); n = UInt(Rn); m = UInt(Rm);
3 if d IN {13,15} || n IN {13,15} || m IN {13,15} then UNPREDICTABLE;

```

### Assembler symbols for all encodings

<c> See [C1.2.5 Standard assembler syntax fields](#) on page 309.  
 <q> See [C1.2.5 Standard assembler syntax fields](#) on page 309.  
 <Rd> Is the general-purpose destination register, encoded in the "Rd" field.  
 <Rn> Is the first general-purpose source register, encoded in the "Rn" field.  
 <Rm> Is the second general-purpose source register, encoded in the "Rm" field.

### Operation for all encodings

```

1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 absdiff1 = Abs(UInt(R[n][7:0]) - UInt(R[m][7:0]));
4 absdiff2 = Abs(UInt(R[n][15:8]) - UInt(R[m][15:8]));
5 absdiff3 = Abs(UInt(R[n][23:16]) - UInt(R[m][23:16]));
6 absdiff4 = Abs(UInt(R[n][31:24]) - UInt(R[m][31:24]));
7 result = absdiff1 + absdiff2 + absdiff3 + absdiff4;
8 R[d] = result[31:0];

```

### C2.4.239 USADA8

Unsigned Sum of Absolute Differences and Accumulate. Unsigned Sum of Absolute Differences and Accumulate performs four unsigned 8-bit subtractions, and adds the absolute values of the differences to a 32-bit accumulate operand.

#### T1

Armv8-M DSP Extension only

|    |    |    |    |    |    |   |   |   |   |   |   |    |   |   |   |            |    |    |    |    |    |   |   |         |   |   |   |    |   |   |   |
|----|----|----|----|----|----|---|---|---|---|---|---|----|---|---|---|------------|----|----|----|----|----|---|---|---------|---|---|---|----|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3  | 2 | 1 | 0 | 15         | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7       | 6 | 5 | 4 | 3  | 2 | 1 | 0 |
| 1  | 1  | 1  | 1  | 1  | 0  | 1 | 1 | 0 | 1 | 1 | 1 | Rn |   |   |   | Ra != 1111 |    |    |    | Rd |    |   |   | 0 0 0 0 |   |   |   | Rm |   |   |   |

#### T1 variant

USADA8{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>

#### Decode for this encoding

```

1 if Ra == '1111' then SEE USAD8;
2 if !HaveDSPExt() then UNDEFINED;
3 d = UInt(Rd); n = UInt(Rn); m = UInt(Rm); a = UInt(Ra);
4 if d IN {13,15} || n IN {13,15} || m IN {13,15} || a == 13 then UNPREDICTABLE;

```

#### Assembler symbols for all encodings

<c> See [C1.2.5 Standard assembler syntax fields](#) on page 309.  
 <q> See [C1.2.5 Standard assembler syntax fields](#) on page 309.  
 <Rd> Is the general-purpose destination register, encoded in the "Rd" field.  
 <Rn> Is the first general-purpose source register, encoded in the "Rn" field.  
 <Rm> Is the second general-purpose source register, encoded in the "Rm" field.  
 <Ra> Is the third general-purpose source register holding the addend, encoded in the "Ra" field.

#### Operation for all encodings

```

1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 absdiff1 = Abs(UInt(R[n][7:0]) - UInt(R[m][7:0]));
4 absdiff2 = Abs(UInt(R[n][15:8]) - UInt(R[m][15:8]));
5 absdiff3 = Abs(UInt(R[n][23:16]) - UInt(R[m][23:16]));
6 absdiff4 = Abs(UInt(R[n][31:24]) - UInt(R[m][31:24]));
7 result = UInt(R[a]) + absdiff1 + absdiff2 + absdiff3 + absdiff4;
8 R[d] = result[31:0];

```

## C2.4.240 USAT

Unsigned Saturate. Unsigned Saturate saturates an optionally-shifted signed value to a selected unsigned range. The Q flag is set to 1 if the operation saturates.

### T1

Armv8-M Main Extension only

|    |    |    |    |    |     |   |   |   |   |    |   |    |   |   |   |    |      |    |    |    |    |   |   |   |      |     |         |   |   |   |   |
|----|----|----|----|----|-----|---|---|---|---|----|---|----|---|---|---|----|------|----|----|----|----|---|---|---|------|-----|---------|---|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10  | 9 | 8 | 7 | 6 | 5  | 4 | 3  | 2 | 1 | 0 | 15 | 14   | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6    | 5   | 4       | 3 | 2 | 1 | 0 |
| 1  | 1  | 1  | 1  | 0  | (0) | 1 | 1 | 1 | 0 | sh | 0 | Rn |   |   |   | 0  | imm3 |    |    |    | Rd |   |   |   | imm2 | (0) | sat_imm |   |   |   |   |

#### Arithmetic shift right variant

Applies when **sh == 1 && !(imm3 == 000 && imm2 == 00)**.

USAT{<c>}{<q>} <Rd>, #<imm>, <Rn>, ASR #<amount>

#### Logical shift left variant

Applies when **sh == 0**.

USAT{<c>}{<q>} <Rd>, #<imm>, <Rn> {, LSL #<amount>}

#### Decode for this encoding

```

1 if sh == '1' && (imm3:imm2) == '0000' then
2 if HaveDSPExt() then
3 SEE USAT16;
4 else
5 UNDEFINED;
6 if !HaveMainExt() then UNDEFINED;
7 d = UInt(Rd); n = UInt(Rn); saturate_to = UInt(sat_imm);
8 (shift_t, shift_n) = DecodeImmShift(sh:'0', imm3:imm2);
9 if d IN {13,15} || n IN {13,15} then UNPREDICTABLE;
```

#### Assembler symbols for all encodings

|          |                                                                                                                                                                                                                                                                                  |
|----------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <c>      | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                                                                                                                                                                                                         |
| <q>      | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                                                                                                                                                                                                         |
| <Rd>     | Is the general-purpose destination register, encoded in the "Rd" field.                                                                                                                                                                                                          |
| <imm>    | Is the bit position for saturation, in the range 0 to 31, encoded in the "sat_imm" field.                                                                                                                                                                                        |
| <Rn>     | Is the general-purpose source register, encoded in the "Rn" field.                                                                                                                                                                                                               |
| <amount> | For the arithmetic shift right variant: is the shift amount, in the range 1 to 31 encoded in the "imm3:imm2" field as <amount>.<br>For the logical shift left variant: is the optional shift amount, in the range 0 to 31, defaulting to 0 and encoded in the "imm3:imm2" field. |

#### Operation for all encodings

```

1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 operand = Shift(R[n], shift_t, shift_n, APSR.C); // APSR.C ignored
4 (result, sat) = UnsignedSatQ(SInt(operand), saturate_to);
5 R[d] = ZeroExtend(result, 32);
6 if sat then
7 APSR.Q = '1';
```

### C2.4.241 USAT16

Unsigned Saturate 16. Unsigned Saturate 16 saturates two signed 16-bit values to a selected unsigned range. The Q flag is set to 1 if the operation saturates.

#### T1

Armv8-M DSP Extension only

|    |    |    |    |    |     |   |   |   |   |   |   |    |   |   |   |    |    |    |    |    |    |   |   |   |   |     |     |         |   |   |   |
|----|----|----|----|----|-----|---|---|---|---|---|---|----|---|---|---|----|----|----|----|----|----|---|---|---|---|-----|-----|---------|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10  | 9 | 8 | 7 | 6 | 5 | 4 | 3  | 2 | 1 | 0 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5   | 4   | 3       | 2 | 1 | 0 |
| 1  | 1  | 1  | 1  | 0  | (0) | 1 | 1 | 1 | 0 | 1 | 0 | Rn |   |   |   | 0  | 0  | 0  | 0  | Rd |    |   |   | 0 | 0 | (0) | (0) | sat_imm |   |   |   |

#### T1 variant

USAT16{<c>}{<q>} <Rd>, #<imm>, <Rn>

#### Decode for this encoding

```

1 if !HaveDSPExt() then UNDEFINED;
2 d = UInt(Rd); n = UInt(Rn); saturate_to = UInt(sat_imm);
3 if d IN {13,15} || n IN {13,15} then UNPREDICTABLE;

```

#### Assembler symbols for all encodings

<c> See [C1.2.5 Standard assembler syntax fields](#) on page 309.  
 <q> See [C1.2.5 Standard assembler syntax fields](#) on page 309.  
 <Rd> Is the general-purpose destination register, encoded in the "Rd" field.  
 <imm> Is the bit position for saturation, in the range 0 to 15, encoded in the "sat\_imm" field.  
 <Rn> Is the general-purpose source register, encoded in the "Rn" field.

#### Operation for all encodings

```

1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 (result1, sat1) = UnsignedSatQ(SInt(R[n][15:0]), saturate_to);
4 (result2, sat2) = UnsignedSatQ(SInt(R[n][31:16]), saturate_to);
5 bits(32) result;
6 result[15:0] = ZeroExtend(result1, 16);
7 result[31:16] = ZeroExtend(result2, 16);
8 R[d] = result;
9 if sat1 || sat2 then
10 APSR.Q = '1';

```

## C2.4.242 USAX

Unsigned Subtract and Add with Exchange. Unsigned Subtract and Add with Exchange exchanges the two halfwords of the second operand, performs one unsigned 16-bit integer subtraction and one unsigned 16-bit addition, and writes the results to the destination register. It sets the APSR.GE bits according to the results.

### T1

Armv8-M DSP Extension only

|    |    |    |    |    |    |   |   |   |   |   |   |    |   |   |   |    |    |    |    |    |    |   |   |   |   |   |   |    |   |   |   |
|----|----|----|----|----|----|---|---|---|---|---|---|----|---|---|---|----|----|----|----|----|----|---|---|---|---|---|---|----|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3  | 2 | 1 | 0 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3  | 2 | 1 | 0 |
| 1  | 1  | 1  | 1  | 1  | 0  | 1 | 0 | 1 | 1 | 1 | 0 | Rn |   |   |   | 1  | 1  | 1  | 1  | Rd |    |   |   | 0 | 1 | 0 | 0 | Rm |   |   |   |

### T1 variant

USAX{<c>} {<q>} {<Rd>}, {<Rn>}, <Rm>

### Decode for this encoding

```

1 if !HaveDSPExt() then UNDEFINED;
2 d = UInt(Rd); n = UInt(Rn); m = UInt(Rm);
3 if d IN {13,15} || n IN {13,15} || m IN {13,15} then UNPREDICTABLE;

```

### Assembler symbols for all encodings

<c> See [C1.2.5 Standard assembler syntax fields](#) on page 309.  
 <q> See [C1.2.5 Standard assembler syntax fields](#) on page 309.  
 <Rd> Is the general-purpose destination register, encoded in the "Rd" field.  
 <Rn> Is the first general-purpose source register, encoded in the "Rn" field.  
 <Rm> Is the second general-purpose source register, encoded in the "Rm" field.

### Operation for all encodings

```

1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 sum = UInt(R[n][15:0]) + UInt(R[m][31:16]);
4 diff = UInt(R[n][31:16]) - UInt(R[m][15:0]);
5 R[d] = diff[15:0] : sum[15:0];
6 APSR.GE[1:0] = if sum >= 0x10000 then '11' else '00';
7 APSR.GE[3:2] = if diff >= 0 then '11' else '00';

```

### C2.4.243 USUB16

Unsigned Subtract 16. Unsigned Subtract 16 performs two 16-bit unsigned integer subtractions, and writes the results to the destination register. It sets the APSR.GE bits according to the results of the subtractions.

#### T1

Armv8-M DSP Extension only

|    |    |    |    |    |    |   |   |   |   |   |   |    |   |   |   |    |    |    |    |    |    |   |   |   |   |   |   |    |   |   |   |
|----|----|----|----|----|----|---|---|---|---|---|---|----|---|---|---|----|----|----|----|----|----|---|---|---|---|---|---|----|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3  | 2 | 1 | 0 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3  | 2 | 1 | 0 |
| 1  | 1  | 1  | 1  | 1  | 0  | 1 | 0 | 1 | 1 | 0 | 1 | Rn |   |   |   | 1  | 1  | 1  | 1  | Rd |    |   |   | 0 | 1 | 0 | 0 | Rm |   |   |   |

#### T1 variant

USUB16{<c>}{<q>} {<Rd>}, {<Rn>}, <Rm>

#### Decode for this encoding

```

1 if !HaveDSPExt() then UNDEFINED;
2 d = UInt(Rd); n = UInt(Rn); m = UInt(Rm);
3 if d IN {13,15} || n IN {13,15} || m IN {13,15} then UNPREDICTABLE;
```

#### Assembler symbols for all encodings

<c> See [C1.2.5 Standard assembler syntax fields](#) on page 309.  
 <q> See [C1.2.5 Standard assembler syntax fields](#) on page 309.  
 <Rd> Is the general-purpose destination register, encoded in the "Rd" field.  
 <Rn> Is the first general-purpose source register, encoded in the "Rn" field.  
 <Rm> Is the second general-purpose source register, encoded in the "Rm" field.

#### Operation for all encodings

```

1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 diff1 = UInt(R[n][15:0]) - UInt(R[m][15:0]);
4 diff2 = UInt(R[n][31:16]) - UInt(R[m][31:16]);
5 R[d] = diff2[15:0] : diff1[15:0];
6 APSR.GE[1:0] = if diff1 >= 0 then '11' else '00';
7 APSR.GE[3:2] = if diff2 >= 0 then '11' else '00';
```

## C2.4.244 USUB8

Unsigned Subtract 8. Unsigned Subtract 8 performs four 8-bit unsigned integer subtractions, and writes the results to the destination register. It sets the APSR.GE bits according to the results of the subtractions.

### T1

Armv8-M DSP Extension only

|    |    |    |    |    |    |   |   |   |   |   |   |    |   |   |   |    |    |    |    |    |    |   |   |   |   |   |   |    |   |   |   |
|----|----|----|----|----|----|---|---|---|---|---|---|----|---|---|---|----|----|----|----|----|----|---|---|---|---|---|---|----|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3  | 2 | 1 | 0 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3  | 2 | 1 | 0 |
| 1  | 1  | 1  | 1  | 1  | 0  | 1 | 0 | 1 | 1 | 0 | 0 | Rn |   |   |   | 1  | 1  | 1  | 1  | Rd |    |   |   | 0 | 1 | 0 | 0 | Rm |   |   |   |

### T1 variant

USUB8 {<c>} {<q>} {<Rd>}, {<Rn>}, {<Rm>}

### Decode for this encoding

```

1 if !HaveDSPExt() then UNDEFINED;
2 d = UInt(Rd); n = UInt(Rn); m = UInt(Rm);
3 if d IN {13,15} || n IN {13,15} || m IN {13,15} then UNPREDICTABLE;

```

### Assembler symbols for all encodings

<c> See [C1.2.5 Standard assembler syntax fields](#) on page 309.  
 <q> See [C1.2.5 Standard assembler syntax fields](#) on page 309.  
 <Rd> Is the general-purpose destination register, encoded in the "Rd" field.  
 <Rn> Is the first general-purpose source register, encoded in the "Rn" field.  
 <Rm> Is the second general-purpose source register, encoded in the "Rm" field.

### Operation for all encodings

```

1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 diff1 = UInt(R[n][7:0]) - UInt(R[m][7:0]);
4 diff2 = UInt(R[n][15:8]) - UInt(R[m][15:8]);
5 diff3 = UInt(R[n][23:16]) - UInt(R[m][23:16]);
6 diff4 = UInt(R[n][31:24]) - UInt(R[m][31:24]);
7 R[d] = diff4[7:0] : diff3[7:0] : diff2[7:0] : diff1[7:0];
8 APSR.GE[0] = if diff1 >= 0 then '1' else '0';
9 APSR.GE[1] = if diff2 >= 0 then '1' else '0';
10 APSR.GE[2] = if diff3 >= 0 then '1' else '0';
11 APSR.GE[3] = if diff4 >= 0 then '1' else '0';

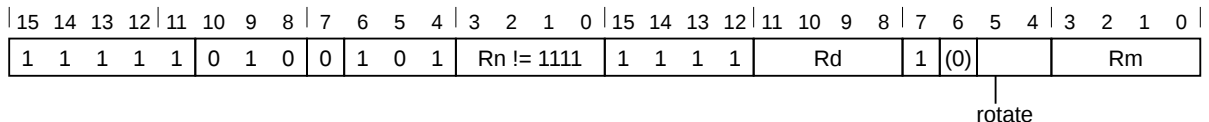
```

## C2.4.245 UXTAB

Unsigned Extend and Add Byte. Unsigned Extend and Add Byte extracts an 8-bit value from a register, zero-extends it to 32 bits, adds the result to the value in another register, and writes the final result to the destination register. The instruction can specify a rotation by 0, 8, 16, or 24 bits before extracting the 8-bit value.

### T1

Armv8-M DSP Extension only



### T1 variant

UXTAB<c>{<q>} {<Rd>,<Rn>,<Rm> {,<ROR #<amount>}}

### Decode for this encoding

```

1 if Rn == '1111' then SEE UXTB;
2 if !HaveDSPExt() then UNDEFINED;
3 d = UInt(Rd); n = UInt(Rn); m = UInt(Rm); rotation = UInt(rotate:'000');
4 if d IN {13,15} || n == 13 || m IN {13,15} then UNPREDICTABLE;

```

### Assembler symbols for all encodings

<c> See [C1.2.5 Standard assembler syntax fields](#) on page 309.  
 <q> See [C1.2.5 Standard assembler syntax fields](#) on page 309.  
 <Rd> Is the general-purpose destination register, encoded in the "Rd" field.  
 <Rn> Is the first general-purpose source register, encoded in the "Rn" field.  
 <Rm> Is the second general-purpose source register, encoded in the "Rm" field.  
 <amount> Is the rotate amount, encoded in the "rotate" field. It can have the following values:  
     0 when rotate = 00  
     8 when rotate = 01  
     16 when rotate = 10  
     24 when rotate = 11  
 ROR #<amount> can be omitted, meaning a rotate amount of 0. This is the preferred disassembly for rotate == 0b00.

### Operation for all encodings

```

1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 rotated = ROR(R[m], rotation);
4 R[d] = R[n] + ZeroExtend(rotated[7:0], 32);

```

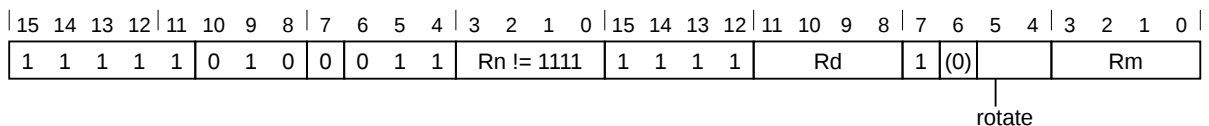


## C2.4.246 UXTAB16

Unsigned Extend and Add Byte 16. Unsigned Extend and Add Byte 16 extracts two 8-bit values from a register, zero-extends them to 16 bits each, adds the results to two 16-bit values from another register, and writes the final results to the destination register. The instruction can specify a rotation by 0, 8, 16, or 24 bits before extracting the 8-bit values.

### T1

Armv8-M DSP Extension only



### T1 variant

UXTAB16{<c>}{<q>} {<Rd>, } <Rn>, <Rm> {, ROR #<amount>}

### Decode for this encoding

```

1 if Rn == '1111' then SEE UXTB16;
2 if !HaveDSPExt() then UNDEFINED;
3 d = UInt(Rd); n = UInt(Rn); m = UInt(Rm); rotation = UInt(rotate:'000');
4 if d IN {13,15} || n == 13 || m IN {13,15} then UNPREDICTABLE;

```

### Assembler symbols for all encodings

|          |                                                                                                                                                                                                                                                                                                                                                                                                                                                          |   |                  |   |                  |    |                  |    |                  |
|----------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---|------------------|---|------------------|----|------------------|----|------------------|
| <c>      | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                                                                                                                                                                                                                                                                                                                                                                                 |   |                  |   |                  |    |                  |    |                  |
| <q>      | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                                                                                                                                                                                                                                                                                                                                                                                 |   |                  |   |                  |    |                  |    |                  |
| <Rd>     | Is the general-purpose destination register, encoded in the "Rd" field.                                                                                                                                                                                                                                                                                                                                                                                  |   |                  |   |                  |    |                  |    |                  |
| <Rn>     | Is the first general-purpose source register, encoded in the "Rn" field.                                                                                                                                                                                                                                                                                                                                                                                 |   |                  |   |                  |    |                  |    |                  |
| <Rm>     | Is the second general-purpose source register, encoded in the "Rm" field.                                                                                                                                                                                                                                                                                                                                                                                |   |                  |   |                  |    |                  |    |                  |
| <amount> | Is the rotate amount, encoded in the "rotate" field. It can have the following values: <table style="margin-left: 20px; border: none;"> <tr><td>0</td><td>when rotate = 00</td></tr> <tr><td>8</td><td>when rotate = 01</td></tr> <tr><td>16</td><td>when rotate = 10</td></tr> <tr><td>24</td><td>when rotate = 11</td></tr> </table> ROR #<amount> can be omitted, meaning a rotate amount of 0. This is the preferred disassembly for rotate == 0b00. | 0 | when rotate = 00 | 8 | when rotate = 01 | 16 | when rotate = 10 | 24 | when rotate = 11 |
| 0        | when rotate = 00                                                                                                                                                                                                                                                                                                                                                                                                                                         |   |                  |   |                  |    |                  |    |                  |
| 8        | when rotate = 01                                                                                                                                                                                                                                                                                                                                                                                                                                         |   |                  |   |                  |    |                  |    |                  |
| 16       | when rotate = 10                                                                                                                                                                                                                                                                                                                                                                                                                                         |   |                  |   |                  |    |                  |    |                  |
| 24       | when rotate = 11                                                                                                                                                                                                                                                                                                                                                                                                                                         |   |                  |   |                  |    |                  |    |                  |

### Operation for all encodings

```

1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 rotated = ROR(R[m], rotation);
4 bits(32) result;
5 result[15:0] = R[n][15:0] + ZeroExtend(rotated[7:0], 16);
6 result[31:16] = R[n][31:16] + ZeroExtend(rotated[23:16], 16);
7 R[d] = result;

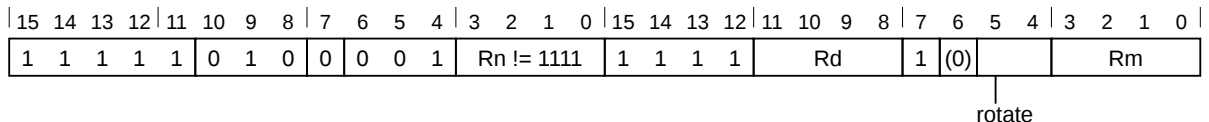
```

## C2.4.247 UXTAH

Unsigned Extend and Add Halfword. Unsigned Extend and Add Halfword extracts a 16-bit value from a register, zero-extends it to 32 bits, adds the result to a value from another register, and writes the final result to the destination register. The instruction can specify a rotation by 0, 8, 16, or 24 bits before extracting the 16-bit value.

### T1

Armv8-M DSP Extension only



### T1 variant

UXTAH<c>{<q>} {<Rd>,<Rn>,<Rm> {,<ROR #<amount>>}

### Decode for this encoding

```

1 if Rn == '1111' then SEE UXTH;
2 if !HaveDSPExt() then UNDEFINED;
3 d = UInt(Rd); n = UInt(Rn); m = UInt(Rm); rotation = UInt(rotate:'000');
4 if d IN {13,15} || n == 13 || m IN {13,15} then UNPREDICTABLE;

```

### Assembler symbols for all encodings

- <c> See [C1.2.5 Standard assembler syntax fields](#) on page 309.
  - <q> See [C1.2.5 Standard assembler syntax fields](#) on page 309.
  - <Rd> Is the general-purpose destination register, encoded in the "Rd" field.
  - <Rn> Is the first general-purpose source register, encoded in the "Rn" field.
  - <Rm> Is the second general-purpose source register, encoded in the "Rm" field.
  - <amount> Is the rotate amount, encoded in the "rotate" field. It can have the following values:
    - 0 when rotate = 00
    - 8 when rotate = 01
    - 16 when rotate = 10
    - 24 when rotate = 11
- ROR #<amount> can be omitted, meaning a rotate amount of 0. This is the preferred disassembly for rotate == 0b00.

### Operation for all encodings

```

1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 rotated = ROR(R[m], rotation);
4 R[d] = R[n] + ZeroExtend(rotated[15:0], 32);

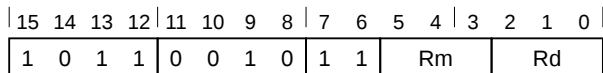
```

### C2.4.248 UXTB

Unsigned Extend Byte. Unsigned Extend Byte extracts an 8-bit value from a register, zero extends it to 32 bits, and writes the result to the destination register. The instruction can specify a rotation by 0, 8, 16, or 24 bits before extracting the 8-bit value.

#### T1

Armv8-M



#### T1 variant

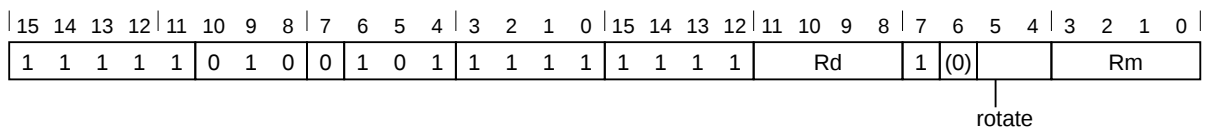
UXTB{<c>}{<q>} {<Rd>}, {<Rm>}

#### Decode for this encoding

```
1 d = UInt(Rd); m = UInt(Rm); rotation = 0;
```

#### T2

Armv8-M Main Extension only



#### T2 variant

UXTB{<c>}.W {<Rd>}, {<Rm>  
 // <Rd>, <Rm> can be represented in T1  
 UXTB{<c>}{<q>} {<Rd>}, {<Rm>} {, ROR #<amount>}

#### Decode for this encoding

```
1 if !HaveMainExt() then UNDEFINED;
2 d = UInt(Rd); m = UInt(Rm); rotation = UInt(rotate: '000');
3 if d IN {13,15} || m IN {13,15} then UNPREDICTABLE;
```

#### Assembler symbols for all encodings

- <c> See [C1.2.5 Standard assembler syntax fields](#) on page 309.
  - <q> See [C1.2.5 Standard assembler syntax fields](#) on page 309.
  - <Rd> Is the general-purpose destination register, encoded in the "Rd" field.
  - <Rm> Is the general-purpose source register, encoded in the "Rm" field.
  - <amount> Is the rotate amount, encoded in the "rotate" field. It can have the following values:
    - 0 when rotate = 00
    - 8 when rotate = 01
    - 16 when rotate = 10
    - 24 when rotate = 11
- ROR #<amount> can be omitted, meaning a rotate amount of 0. This is the preferred disassembly for rotate == 0b00.

### Operation for all encodings

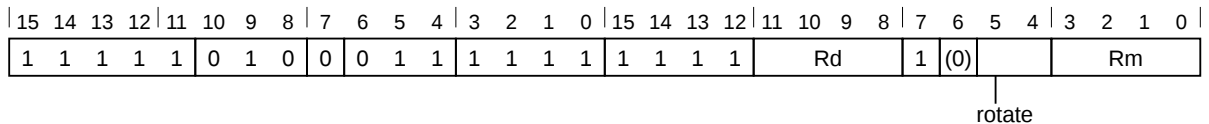
```
1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 rotated = ROR(R[m], rotation);
4 R[d] = ZeroExtend(rotated[7:0], 32);
```

### C2.4.249 UXTB16

Unsigned Extend Byte 16. Unsigned Extend Byte 16 extracts two 8-bit values from a register, zero-extends them to 16 bits each, and writes the results to the destination register. The instruction can specify a rotation by 0, 8, 16, or 24 bits before extracting the 8-bit values.

#### T1

Armv8-M DSP Extension only



#### T1 variant

UXTB16{<c>}{<q>} {<Rd>}, <Rm> {, ROR #<amount>}

#### Decode for this encoding

```

1 if !HaveDSPExt() then UNDEFINED;
2 d = UInt(Rd); m = UInt(Rm); rotation = UInt(rotate: '000');
3 if d IN {13,15} || m IN {13,15} then UNPREDICTABLE;

```

#### Assembler symbols for all encodings

- <c> See [C1.2.5 Standard assembler syntax fields](#) on page 309.
- <q> See [C1.2.5 Standard assembler syntax fields](#) on page 309.
- <Rd> Is the general-purpose destination register, encoded in the "Rd" field.
- <Rm> Is the second general-purpose source register, encoded in the "Rm" field.
- <amount> Is the rotate amount, encoded in the "rotate" field. It can have the following values:
  - 0 when rotate = 00
  - 8 when rotate = 01
  - 16 when rotate = 10
  - 24 when rotate = 11
 ROR #<amount> can be omitted, meaning a rotate amount of 0. This is the preferred disassembly for rotate == 0b00.

#### Operation for all encodings

```

1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 rotated = ROR(R[m], rotation);
4 bits(32) result;
5 result[15:0] = ZeroExtend(rotated[7:0], 16);
6 result[31:16] = ZeroExtend(rotated[23:16], 16);
7 R[d] = result;

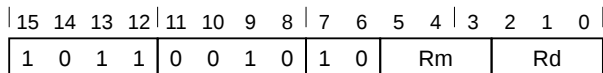
```

### C2.4.250 UXTH

Unsigned Extend Halfword. Unsigned Extend Halfword extracts a 16-bit value from a register, zero extends it to 32 bits, and writes the result to the destination register. The instruction can specify a rotation by 0, 8, 16, or 24 bits before extracting the 16-bit value.

#### T1

Armv8-M



#### T1 variant

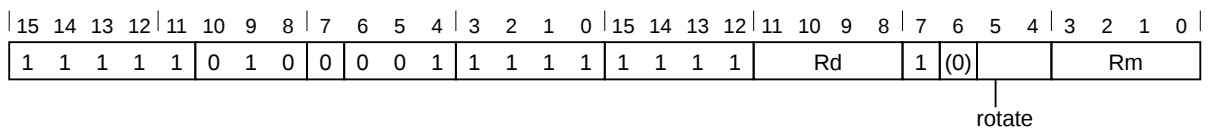
UXTH{<c>}{<q>} {<Rd>}, {<Rm>}

#### Decode for this encoding

```
1 d = UInt(Rd); m = UInt(Rm); rotation = 0;
```

#### T2

Armv8-M Main Extension only



#### T2 variant

UXTH{<c>}.W {<Rd>}, {<Rm>  
 // <Rd>, <Rm> can be represented in T1  
 UXTH{<c>}{<q>} {<Rd>}, {<Rm> {, ROR #<amount>}}

#### Decode for this encoding

```
1 if !HaveMainExt() then UNDEFINED;
2 d = UInt(Rd); m = UInt(Rm); rotation = UInt(rotate: '000');
3 if d IN {13,15} || m IN {13,15} then UNPREDICTABLE;
```

#### Assembler symbols for all encodings

- <c> See [C1.2.5 Standard assembler syntax fields](#) on page 309.
- <q> See [C1.2.5 Standard assembler syntax fields](#) on page 309.
- <Rd> Is the general-purpose destination register, encoded in the "Rd" field.
- <Rm> Is the general-purpose source register, encoded in the "Rm" field.
- <amount> Is the rotate amount, encoded in the "rotate" field. It can have the following values:
  - 0 when rotate = 00
  - 8 when rotate = 01
  - 16 when rotate = 10
  - 24 when rotate = 11
 ROR #<amount> can be omitted, meaning a rotate amount of 0. This is the preferred disassembly for rotate == 0b00.

### Operation for all encodings

```
1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 rotated = ROR(R[m], rotation);
4 R[d] = ZeroExtend(rotated[15:0], 32);
```

## C2.4.251 VABS

Floating-point Absolute. Floating-point Absolute takes the absolute value of a single-precision or double-precision register, and places the result in the destination register.

### T2

*Armv8-M Floating-point Extension only*, *sz == 1 UNDEFINED in single-precision only implementations.*

|    |    |    |    |    |    |   |   |   |   |   |   |   |   |   |   |    |    |    |    |    |    |   |   |   |   |   |   |   |   |   |    |
|----|----|----|----|----|----|---|---|---|---|---|---|---|---|---|---|----|----|----|----|----|----|---|---|---|---|---|---|---|---|---|----|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0  |
| 1  | 1  | 1  | 0  | 1  | 1  | 1 | 0 | 1 | D | 1 | 1 | 0 | 0 | 0 | 0 | Vd | 1  | 0  | 1  | sz | 1  | 1 | M | 0 |   |   |   |   |   |   | Vm |

#### Single-precision scalar variant

*Armv8-M Floating-point Extension only.*

Applies when **sz == 0**.

VABS{<c>}{<q>}.F32 <Sd>, <Sm>

#### Double-precision scalar variant

*Armv8-M Floating-point Extension only.*

Applies when **sz == 1**.

VABS{<c>}{<q>}.F64 <Dd>, <Dm>

#### Decode for this encoding

```

1 dp_operation = (sz == '1');
2 CheckDecodeFaults(dp_operation);
3 d = if dp_operation then UInt(D:Vd) else UInt(Vd:D);
4 m = if dp_operation then UInt(M:Vm) else UInt(Vm:M);

```

#### Assembler symbols for all encodings

|      |                                                                                             |
|------|---------------------------------------------------------------------------------------------|
| <c>  | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                    |
| <q>  | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                    |
| <Sd> | Is the 32-bit name of the floating-point destination register, encoded in the "Vd:D" field. |
| <Sm> | Is the 32-bit name of the floating-point source register, encoded in the "Vm:M" field.      |
| <Dd> | Is the 64-bit name of the floating-point destination register, encoded in the "D:Vd" field. |
| <Dm> | Is the 64-bit name of the floating-point source register, encoded in the "M:Vm" field.      |

#### Operation for all encodings

```

1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 ExecuteFPCheck();
4 if dp_operation then
5 D[d] = FPAbs(D[m]);
6 else
7 S[d] = FPAbs(S[m]);

```



## C2.4.252 VADD

Floating-point Add. Floating-point Add adds two single-precision or double-precision registers, and places the result in the destination register.

### T2

*Armv8-M Floating-point Extension only*, *sz == 1* UNDEFINED in single-precision only implementations.

|    |    |    |    |    |    |   |   |   |   |   |   |    |   |   |   |    |    |    |    |    |    |   |    |   |   |   |   |    |   |   |   |
|----|----|----|----|----|----|---|---|---|---|---|---|----|---|---|---|----|----|----|----|----|----|---|----|---|---|---|---|----|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3  | 2 | 1 | 0 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8  | 7 | 6 | 5 | 4 | 3  | 2 | 1 | 0 |
| 1  | 1  | 1  | 0  | 1  | 1  | 1 | 0 | 0 | D | 1 | 1 | Vn |   |   |   | Vd |    |    |    | 1  | 0  | 1 | sz | N | 0 | M | 0 | Vm |   |   |   |

#### Single-precision scalar variant

*Armv8-M Floating-point Extension only*.

Applies when **sz == 0**.

VADD{<c>}{<q>}.F32 {<Sd>}, {<Sn>}, {<Sm>}

#### Double-precision scalar variant

*Armv8-M Floating-point Extension only*.

Applies when **sz == 1**.

VADD{<c>}{<q>}.F64 {<Dd>}, {<Dn>}, {<Dm>}

#### Decode for this encoding

```

1 dp_operation = (sz == '1');
2 CheckDecodeFaults(dp_operation);
3 d = if dp_operation then UInt(D:Vd) else UInt(Vd:D);
4 n = if dp_operation then UInt(N:Vn) else UInt(Vn:N);
5 m = if dp_operation then UInt(M:Vm) else UInt(Vm:M);

```

#### Assembler symbols for all encodings

|      |                                                                                               |
|------|-----------------------------------------------------------------------------------------------|
| <c>  | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                      |
| <q>  | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                      |
| <Sd> | Is the 32-bit name of the floating-point destination register, encoded in the "Vd:D" field.   |
| <Sn> | Is the 32-bit name of the first floating-point source register, encoded in the "Vn:N" field.  |
| <Sm> | Is the 32-bit name of the second floating-point source register, encoded in the "Vm:M" field. |
| <Dd> | Is the 64-bit name of the floating-point destination register, encoded in the "D:Vd" field.   |
| <Dn> | Is the 64-bit name of the first floating-point source register, encoded in the "N:Vn" field.  |
| <Dm> | Is the 64-bit name of the second floating-point source register, encoded in the "M:Vm" field. |

#### Operation for all encodings

```

1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 ExecuteFPCheck();
4 if dp_operation then
5 D[d] = FPAdd(D[n], D[m], TRUE);
6 else
7 S[d] = FPAdd(S[n], S[m], TRUE);

```

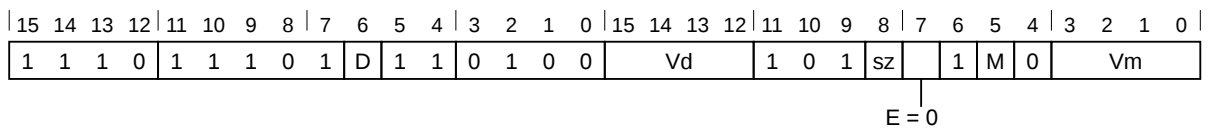
### C2.4.253 VCMF

Floating-point Compare. Floating-point Compare compares two registers, or one register and zero. It writes the result to FPSCR condition flags. These are normally transferred to the APSR condition flags by a subsequent [VMRS](#) instruction.

It raises an Invalid Operation exception only if either operand is a signaling NaN.

#### T1

*Armv8-M Floating-point Extension only, sz == 1 UNDEFINED in single-precision only implementations.*



#### Single-precision scalar variant

*Armv8-M Floating-point Extension only.*

Applies when **sz == 0**.

VCMF {<c>} {<q>}.F32 <Sd>, <Sm>

#### Double-precision scalar variant

*Armv8-M Floating-point Extension only.*

Applies when **sz == 1**.

VCMF {<c>} {<q>}.F64 <Dd>, <Dm>

#### Decode for this encoding

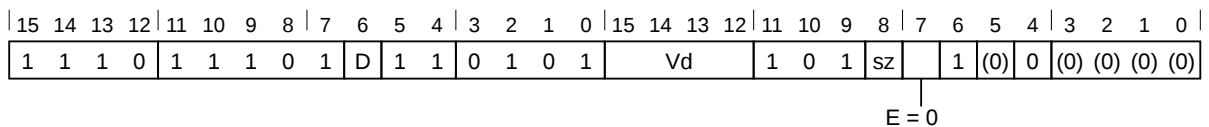
```

1 dp_operation = (sz == '1');
2 CheckDecodeFaults(dp_operation);
3 quiet_nan_exc = (E == '1'); with_zero = FALSE;
4 d = if dp_operation then UInt(D:Vd) else UInt(Vd:D);
5 m = if dp_operation then UInt(M:Vm) else UInt(Vm:M);

```

#### T2

*Armv8-M Floating-point Extension only, sz == 1 UNDEFINED in single-precision only implementations.*



#### Single-precision scalar variant

*Armv8-M Floating-point Extension only.*

Applies when **sz == 0**.

VCMF {<c>} {<q>}.F32 <Sd>, #0.0

#### Double-precision scalar variant

*Armv8-M Floating-point Extension only.*

Applies when **sz == 1**.

```
VCMP{<c>}{<q>}.F64 <Dd>, #0.0
```

### Decode for this encoding

```
1 dp_operation = (sz == '1');
2 CheckDecodeFaults(dp_operation);
3 quiet_nan_exc = (E == '1'); with_zero = TRUE;
4 d = if dp_operation then UInt(D:Vd) else UInt(Vd:D);
5 m = integer UNKNOWN;
```

### Assembler symbols for all encodings

<c> See [C1.2.5 Standard assembler syntax fields](#) on page 309.

<q> See [C1.2.5 Standard assembler syntax fields](#) on page 309.

<Sd> Is the 32-bit name of the floating-point destination register, encoded in the "Vd:D" field.

<Sm> Is the 32-bit name of the floating-point source register, encoded in the "Vm:M" field.

<Dd> Is the 64-bit name of the floating-point destination register, encoded in the "D:Vd" field.

<Dm> Is the 64-bit name of the floating-point source register, encoded in the "M:Vm" field.

### Operation for all encodings

```
1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 ExecuteFPCheck();
4 if dp_operation then
5 op64 = if with_zero then FPZero('0',64) else D[m];
6 (FPSCR.N, FPSCR.Z, FPSCR.C, FPSCR.V) = FPCompare(D[d], op64, quiet_nan_exc, TRUE);
7 else
8 op32 = if with_zero then FPZero('0',32) else S[m];
9 (FPSCR.N, FPSCR.Z, FPSCR.C, FPSCR.V) = FPCompare(S[d], op32, quiet_nan_exc, TRUE);
```

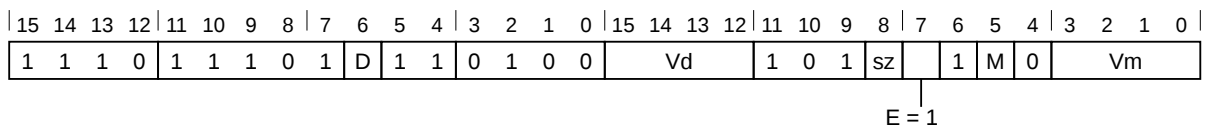
### C2.4.254 VCMPE

Floating-point Compare, raising Invalid Operation on NaN. Floating-point Compare, raising Invalid Operation on NaN compares two registers, or one register and zero. It writes the result to FPSCR condition flags. These are normally transferred to the APSR condition flags by a subsequent **VMRS** instruction.

It raises an Invalid Operation exception if either operand is any type of NaN.

#### T1

*Armv8-M Floating-point Extension only*, *sz == 1* UNDEFINED in single-precision only implementations.



#### Single-precision scalar variant

*Armv8-M Floating-point Extension only*.

Applies when **sz == 0**.

VCMPE{<c>}{<q>}.F32 <Sd>, <Sm>

#### Double-precision scalar variant

*Armv8-M Floating-point Extension only*.

Applies when **sz == 1**.

VCMPE{<c>}{<q>}.F64 <Dd>, <Dm>

#### Decode for this encoding

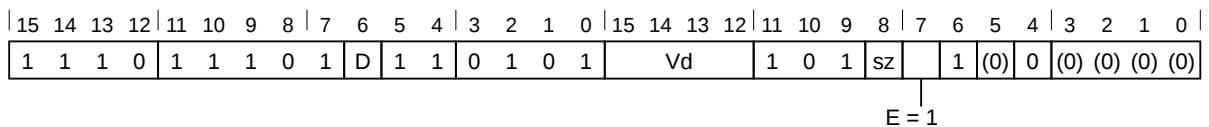
```

1 dp_operation = (sz == '1');
2 CheckDecodeFaults(dp_operation);
3 quiet_nan_exc = (E == '1'); with_zero = FALSE;
4 d = if dp_operation then UInt(D:Vd) else UInt(Vd:D);
5 m = if dp_operation then UInt(M:Vm) else UInt(Vm:M);

```

#### T2

*Armv8-M Floating-point Extension only*, *sz == 1* UNDEFINED in single-precision only implementations.



#### Single-precision scalar variant

*Armv8-M Floating-point Extension only*.

Applies when **sz == 0**.

VCMPE{<c>}{<q>}.F32 <Sd>, #0.0

#### Double-precision scalar variant

*Armv8-M Floating-point Extension only*.

Applies when **sz == 1**.

VCMPE{<c>}{<q>}.F64 <Dd>, #0.0

### Decode for this encoding

```
1 dp_operation = (sz == '1');
2 CheckDecodeFaults(dp_operation);
3 quiet_nan_exc = (E == '1'); with_zero = TRUE;
4 d = if dp_operation then UInt(D:Vd) else UInt(Vd:D);
5 m = integer UNKNOWN;
```

### Assembler symbols for all encodings

<c> See [C1.2.5 Standard assembler syntax fields](#) on page 309.  
<q> See [C1.2.5 Standard assembler syntax fields](#) on page 309.  
<Sd> Is the 32-bit name of the floating-point destination register, encoded in the "Vd:D" field.  
<Sm> Is the 32-bit name of the floating-point source register, encoded in the "Vm:M" field.  
<Dd> Is the 64-bit name of the floating-point destination register, encoded in the "D:Vd" field.  
<Dm> Is the 64-bit name of the floating-point source register, encoded in the "M:Vm" field.

### Operation for all encodings

```
1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 ExecuteFPCheck();
4 if dp_operation then
5 op64 = if with_zero then FPZero('0',64) else D[m];
6 (FPSCR.N, FPSCR.Z, FPSCR.C, FPSCR.V) = FPCompare(D[d], op64, quiet_nan_exc, TRUE);
7 else
8 op32 = if with_zero then FPZero('0',32) else S[m];
9 (FPSCR.N, FPSCR.Z, FPSCR.C, FPSCR.V) = FPCompare(S[d], op32, quiet_nan_exc, TRUE);
```

### C2.4.255 VCVT (between double-precision and single-precision)

Convert between double-precision and single-precision. This instruction does one of the following:

- Converts the value in a double-precision register to single-precision and writes the result to a single-precision register.
- Converts the value in a single-precision register to double-precision and writes the result to a double-precision register.

#### T1

Armv8-M Floating-point Extension only

|    |    |    |    |    |    |   |   |   |   |   |   |   |   |   |   |    |    |    |    |    |    |   |    |   |   |   |   |    |   |   |   |
|----|----|----|----|----|----|---|---|---|---|---|---|---|---|---|---|----|----|----|----|----|----|---|----|---|---|---|---|----|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8  | 7 | 6 | 5 | 4 | 3  | 2 | 1 | 0 |
| 1  | 1  | 1  | 0  | 1  | 1  | 1 | 0 | 1 | D | 1 | 1 | 0 | 1 | 1 | 1 | Vd |    |    |    | 1  | 0  | 1 | sz | 1 | 1 | M | 0 | Vm |   |   |   |

#### Encoding

Applies when **sz == 0**.

VCVT{<c>}{<q>}.F64.F32 <Dd>, <Sm>

#### Encoding

Applies when **sz == 1**.

VCVT{<c>}{<q>}.F32.F64 <Sd>, <Dm>

#### Decode for this encoding

```

1 CheckDecodeFaults(TRUE);
2 double_to_single = (sz == '1');
3 d = if double_to_single then UInt(Vd:D) else UInt(D:Vd);
4 m = if double_to_single then UInt(M:Vm) else UInt(Vm:M);

```

#### Assembler symbols for all encodings

<c> See [C1.2.5 Standard assembler syntax fields](#) on page 309.  
 <q> See [C1.2.5 Standard assembler syntax fields](#) on page 309.  
 <Sd> Is the 32-bit name of the floating-point destination register, encoded in the "Vd:D" field.  
 <Dm> Is the 64-bit name of the floating-point source register, encoded in the "M:Vm" field.  
 <Dd> Is the 64-bit name of the floating-point destination register, encoded in the "D:Vd" field.  
 <Sm> Is the 32-bit name of the floating-point source register, encoded in the "Vm:M" field.

#### Operation for all encodings

```

1 if ConditionPassed() then
2 EncodingSpecificOperations(); ExecuteFPCheck();
3 if double_to_single then
4 S[d] = FPDoubleToSingle(D[m], TRUE);
5 else
6 D[d] = FPSingleToDouble(S[m], TRUE);

```

### C2.4.256 VCVT (between floating-point and fixed-point)

Floating-point Convert (between floating-point and fixed-point). Floating-point Convert (between floating-point and fixed-point) converts a value in a register from floating-point to fixed-point, or from fixed-point to floating-point, and places the result in the destination register. Software can specify the fixed-point value as either signed or unsigned.

The fixed-point value can be 16-bit or 32-bit. Conversions from fixed-point values take their operand from the low-order bits of the source register and ignore any remaining bits. Signed conversions to fixed-point values sign-extend the result value to the destination register width. Unsigned conversions to fixed-point values zero-extend the result value to the destination register width.

The floating-point to fixed-point operation uses the Round towards Zero rounding mode. The fixed-point to floating-point operation uses the Round to Nearest rounding mode.

#### T1

*Armv8-M Floating-point Extension only*, `sf == 1` UNDEFINED in single-precision only implementations.

|    |    |    |    |    |    |   |   |   |   |   |   |   |    |   |   |    |    |    |    |    |    |   |    |    |   |   |   |      |   |   |   |
|----|----|----|----|----|----|---|---|---|---|---|---|---|----|---|---|----|----|----|----|----|----|---|----|----|---|---|---|------|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2  | 1 | 0 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8  | 7  | 6 | 5 | 4 | 3    | 2 | 1 | 0 |
| 1  | 1  | 1  | 0  | 1  | 1  | 1 | 0 | 1 | D | 1 | 1 | 1 | op | 1 | U | Vd |    |    |    | 1  | 0  | 1 | sf | sx | 1 | i | 0 | imm4 |   |   |   |

#### Single-precision scalar variant

*Armv8-M Floating-point Extension only.*

Applies when `op == 0` && `sf == 0`.

`VCVT{<c>}{<q>}.F32.<dt> <Sdm>, <Sdm>, #<fbits>`

#### Single-precision scalar variant

*Armv8-M Floating-point Extension only.*

Applies when `op == 1` && `sf == 0`.

`VCVT{<c>}{<q>}.<dt>.F32 <Sdm>, <Sdm>, #<fbits>`

#### Double-precision scalar variant

*Armv8-M Floating-point Extension only.*

Applies when `op == 0` && `sf == 1`.

`VCVT{<c>}{<q>}.F64.<dt> <Ddm>, <Ddm>, #<fbits>`

#### Double-precision scalar variant

*Armv8-M Floating-point Extension only.*

Applies when `op == 1` && `sf == 1`.

`VCVT{<c>}{<q>}.<dt>.F64 <Ddm>, <Ddm>, #<fbits>`

#### Decode for this encoding

```

1 dp_operation = (sf == '1');
2 CheckDecodeFaults(dp_operation);
3 to_fixed = (op == '1'); unsigned = (U == '1');
4 size = if sx == '0' then 16 else 32;
5 frac_bits = size - UInt(imm4:i);
6 if to_fixed then
7 round_zero = TRUE;
```

```

8 else
9 round_nearest = TRUE;
10 d = if dp_operation then UInt(D:Vd) else UInt(Vd:D);
11 if frac_bits < 0 then UNPREDICTABLE;

```

### CONSTRAINED UNPREDICTABLE behavior

If `frac_bits < 0`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The value in the destination register is UNKNOWN.

### Assembler symbols for all encodings

|         |                                                                                                                                                                                                                                                                     |
|---------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <c>     | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                                                                                                                                                                                            |
| <q>     | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                                                                                                                                                                                            |
| <dt>    | Is the data type for the fixed-point number, encoded in the "U:sx" field. It can have the following values:<br>S16 when U = 0, sx = 0<br>S32 when U = 0, sx = 1<br>U16 when U = 1, sx = 0<br>U32 when U = 1, sx = 1                                                 |
| <Sdm>   | Is the 32-bit name of the floating-point destination and source register, encoded in the "Vd:D" field.                                                                                                                                                              |
| <Ddm>   | Is the 64-bit name of the floating-point destination and source register, encoded in the "D:Vd" field.                                                                                                                                                              |
| <fbits> | The number of fraction bits in the fixed-point number:<br>- If <dt> is S16 or U16, <fbits> must be in the range 0-16. (16 - <fbits>) is encoded in [imm4, i]<br>- If <dt> is S32 or U32, <fbits> must be in the range 1-32. (32 - <fbits>) is encoded in [imm4, i]. |

### Operation for all encodings

```

1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 ExecuteFPCheck();
4 if to_fixed then
5 if dp_operation then
6 result = FPToFixed(D[d], size, frac_bits, unsigned, round_zero, TRUE);
7 D[d] = if unsigned then ZeroExtend(result, 64) else SignExtend(result, 64);
8 else
9 result = FPToFixed(S[d], size, frac_bits, unsigned, round_zero, TRUE);
10 S[d] = if unsigned then ZeroExtend(result, 32) else SignExtend(result, 32);
11 else
12 if dp_operation then
13 D[d] = FixedToFP(D[d][size-1:0], 64, frac_bits, unsigned, round_nearest, TRUE);
14 else
15 S[d] = FixedToFP(S[d][size-1:0], 32, frac_bits, unsigned, round_nearest, TRUE);

```

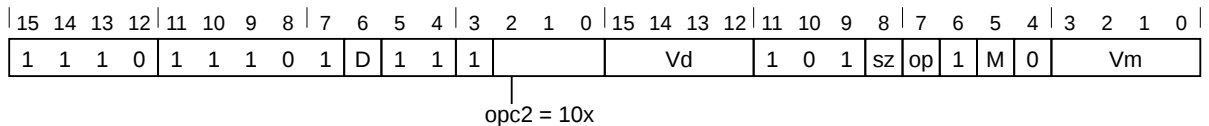


### C2.4.257 VCVT (floating-point to integer)

Convert floating-point to integer with Round towards Zero. Convert floating-point to integer with Round towards Zero converts a value in a register from floating-point to a 32-bit integer, using the Round towards Zero rounding mode, and places the result in the destination register.

#### T1

*Armv8-M Floating-point Extension only*, `sz == 1` UNDEFINED in single-precision only implementations.



#### Single-precision scalar variant

*Armv8-M Floating-point Extension only*.

Applies when `opc2 == 100` && `sz == 0` && `op == 1`.

`VCVT{<c>}{<q>}.U32.F32 <Sd>, <Sm>`

#### Single-precision scalar variant

*Armv8-M Floating-point Extension only*.

Applies when `opc2 == 101` && `sz == 0` && `op == 1`.

`VCVT{<c>}{<q>}.S32.F32 <Sd>, <Sm>`

#### Double-precision scalar variant

*Armv8-M Floating-point Extension only*.

Applies when `opc2 == 100` && `sz == 1` && `op == 1`.

`VCVT{<c>}{<q>}.U32.F64 <Sd>, <Dm>`

#### Double-precision scalar variant

*Armv8-M Floating-point Extension only*.

Applies when `opc2 == 101` && `sz == 1` && `op == 1`.

`VCVT{<c>}{<q>}.S32.F64 <Sd>, <Dm>`

#### Decode for this encoding

```

1 if opc2 != '000' && !(opc2 IN '10x') then SEE "Related encodings";
2 dp_operation = (sz == '1');
3 CheckDecodeFaults(dp_operation);
4 to_integer = (opc2[2] == '1');
5 if to_integer then
6 unsigned = (opc2[0] == '0'); round_zero = (op == '1');
7 d = UInt(Vd:D); m = if dp_operation then UInt(M:Vm) else UInt(Vm:M);
8 else
9 unsigned = (op == '0'); round_nearest = FALSE; // FALSE selects FPSCR rounding
10 m = UInt(Vm:M); d = if dp_operation then UInt(D:Vd) else UInt(Vd:D);

```

#### Assembler symbols for all encodings

<c> See [C1.2.5 Standard assembler syntax fields](#) on page 309.

|      |                                                                                             |
|------|---------------------------------------------------------------------------------------------|
| <q>  | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                    |
| <Sd> | Is the 32-bit name of the floating-point destination register, encoded in the "Vd:D" field. |
| <Sm> | Is the 32-bit name of the floating-point source register, encoded in the "Vm:M" field.      |
| <Dm> | Is the 64-bit name of the floating-point source register, encoded in the "M:Vm" field.      |

### Operation for all encodings

```

1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 ExecuteFPCheck();
4 if to_integer then
5 if dp_operation then
6 S[d] = FPToFixed(D[m], 32, 0, unsigned, round_zero, TRUE);
7 else
8 S[d] = FPToFixed(S[m], 32, 0, unsigned, round_zero, TRUE);
9 else
10 if dp_operation then
11 D[d] = FixedToFP(S[m], 64, 0, unsigned, round_nearest, TRUE);
12 else
13 S[d] = FixedToFP(S[m], 32, 0, unsigned, round_nearest, TRUE);

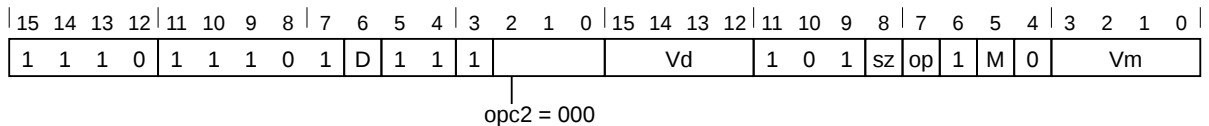
```

## C2.4.258 VCVT (integer to floating-point)

Convert integer to floating-point. Convert integer to floating-point converts a value in a register from a 32-bit integer to floating-point, using the rounding mode specified by FPSCR, and places the result in the destination register.

### T1

*Armv8-M Floating-point Extension only*, `sz == 1` UNDEFINED in single-precision only implementations.



### Single-precision scalar variant

*Armv8-M Floating-point Extension only*.

Applies when `sz == 0`.

`VCVT{<c>}{<q>}.F32.<dt> <Sd>, <Sm>`

### Double-precision scalar variant

*Armv8-M Floating-point Extension only*.

Applies when `sz == 1`.

`VCVT{<c>}{<q>}.F64.<dt> <Dd>, <Sm>`

### Decode for this encoding

```

1 if opc2 != '000' && !(opc2 IN '10x') then SEE "Related encodings";
2 dp_operation = (sz == '1');
3 CheckDecodeFaults(dp_operation);
4 to_integer = (opc2[2] == '1');
5 if to_integer then
6 unsigned = (opc2[0] == '0'); round_zero = (op == '1');
7 d = UInt(Vd:D); m = if dp_operation then UInt(M:Vm) else UInt(Vm:M);
8 else
9 unsigned = (op == '0'); round_nearest = FALSE; // FALSE selects FPSCR rounding
10 m = UInt(Vm:M); d = if dp_operation then UInt(D:Vd) else UInt(Vd:D);

```

### Assembler symbols for all encodings

- <c> See [C1.2.5 Standard assembler syntax fields](#) on page 309.
- <q> See [C1.2.5 Standard assembler syntax fields](#) on page 309.
- <dt> Is the data type for the operand, encoded in the "op" field. It can have the following values:  
     U32 when op = 0  
     S32 when op = 1
- <Sd> Is the 32-bit name of the floating-point destination register, encoded in the "Vd:D" field.
- <Dd> Is the 64-bit name of the floating-point destination register, encoded in the "D:Vd" field.
- <Sm> Is the 32-bit name of the floating-point source register, encoded in the "Vm:M" field.

### Operation for all encodings

```

1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 ExecuteFPCheck();
4 if to_integer then

```

```
5 if dp_operation then
6 S[d] = FPToFixed(D[m], 32, 0, unsigned, round_zero, TRUE);
7 else
8 S[d] = FPToFixed(S[m], 32, 0, unsigned, round_zero, TRUE);
9 else
10 if dp_operation then
11 D[d] = FixedToFP(S[m], 64, 0, unsigned, round_nearest, TRUE);
12 else
13 S[d] = FixedToFP(S[m], 32, 0, unsigned, round_nearest, TRUE);
```

## C2.4.259 VCVTA

Convert floating-point to integer with Round to Nearest with Ties to Away. Convert floating-point to integer with Round to Nearest with Ties to Away converts a value in a register from floating-point to a 32-bit integer using the Round to Nearest with Ties to Away rounding mode, and places the result in the destination register.

### T1

*Armv8-M Floating-point Extension only*,  $sz == 1$  UNDEFINED in single-precision only implementations.

|    |    |    |    |    |    |   |   |   |   |   |   |   |   |         |   |    |    |    |    |    |    |   |    |    |   |   |   |   |   |    |   |
|----|----|----|----|----|----|---|---|---|---|---|---|---|---|---------|---|----|----|----|----|----|----|---|----|----|---|---|---|---|---|----|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1       | 0 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8  | 7  | 6 | 5 | 4 | 3 | 2 | 1  | 0 |
| 1  | 1  | 1  | 1  | 1  | 1  | 1 | 0 | 1 | D | 1 | 1 | 1 | 1 | RM = 00 |   | Vd |    |    |    | 1  | 0  | 1 | sz | op | 1 | M | 0 |   |   | Vm |   |

### Single-precision scalar variant

*Armv8-M Floating-point Extension only*.

Applies when  $sz == 0$ .

VCVTA{<q>}.<dt>.F32 <Sd>, <Sm>

### Double-precision scalar variant

*Armv8-M Floating-point Extension only*.

Applies when  $sz == 1$ .

VCVTA{<q>}.<dt>.F64 <Sd>, <Dm>

### Decode for this encoding

```

1 dp_operation = (sz == '1');
2 CheckDecodeFaults(dp_operation);
3 if InITBlock() then UNPREDICTABLE;
4 unsigned = (op == '0');
5 round_mode = RM;
6 d = UInt(Vd:D);
7 m = if dp_operation then UInt(M:Vm) else UInt(Vm:M);

```

### Assembler symbols for all encodings

<q> See [C1.2.5 Standard assembler syntax fields](#) on page 309.

<dt> Is the data type for the elements of the destination, encoded in the "op" field. It can have the following values:  
 U32 when op = 0  
 S32 when op = 1

<Sd> Is the 32-bit name of the floating-point destination register, encoded in the "Vd:D" field.

<Sm> Is the 32-bit name of the floating-point source register, encoded in the "Vm:M" field.

<Dm> Is the 64-bit name of the floating-point source register, encoded in the "M:Vm" field.

### Operation for all encodings

```

1 EncodingSpecificOperations();
2 ExecuteFPCheck();
3
4 if dp_operation then
5 S[d] = FPToFixedDirected(D[m], 0, unsigned, round_mode, TRUE);
6 else
7 S[d] = FPToFixedDirected(S[m], 0, unsigned, round_mode, TRUE);

```

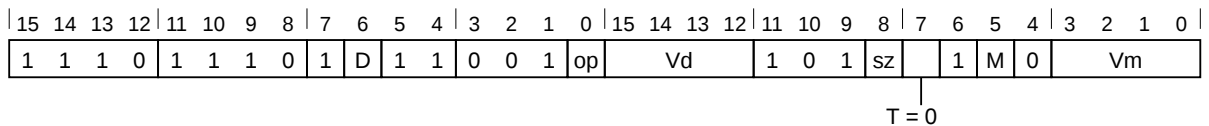
## C2.4.260 VCVTB

Floating-point Convert Bottom. Floating-point Convert Bottom does one of the following:

- Converts the half-precision value in the bottom half of a single-precision register to single-precision and writes the result to a single-precision register.
- Converts the value in a single-precision register to half-precision and writes the result into the bottom half of a single-precision register, preserving the other half of the target register.
- Converts the half-precision value in the bottom half of a single-precision register to double-precision and writes the result to a double-precision register, without intermediate rounding.
- Converts the value in the double-precision register to half-precision and writes the result into the bottom half of a single-precision register, preserving the other half of the target register, without intermediate rounding.

### T1

*Armv8-M Floating-point Extension only*,  $sz == 1$  UNDEFINED in single-precision only implementations.



#### Single-precision scalar variant

*Armv8-M Floating-point Extension only.*

Applies when  $op == 0$  &&  $sz == 0$ .

`VCVTB{<c>}{<q>}.F32.F16 <Sd>, <Sm>`

#### Single-precision scalar variant

*Armv8-M Floating-point Extension only.*

Applies when  $op == 1$  &&  $sz == 0$ .

`VCVTB{<c>}{<q>}.F16.F32 <Sd>, <Sm>`

#### Double-precision scalar variant

*Armv8-M Floating-point Extension only.*

Applies when  $op == 0$  &&  $sz == 1$ .

`VCVTB{<c>}{<q>}.F64.F16 <Dd>, <Sm>`

#### Double-precision scalar variant

*Armv8-M Floating-point Extension only.*

Applies when  $op == 1$  &&  $sz == 1$ .

`VCVTB{<c>}{<q>}.F16.F64 <Sd>, <Dm>`

### Decode for this encoding

```

1 dp_operation = (sz == '1');
2 CheckDecodeFaults(dp_operation);
3 convert_from_half = (op == '0');
4 lowbit = if T == '1' then 16 else 0;
5 if dp_operation then

```

```

6 if convert_from_half then
7 d = UInt(D:Vd); m = UInt(Vm:M);
8 else
9 d = UInt(Vd:D); m = UInt(M:Vm);
10 else
11 d = UInt(Vd:D); m = UInt(Vm:M);

```

### Assembler symbols for all encodings

|      |                                                                                             |
|------|---------------------------------------------------------------------------------------------|
| <c>  | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                    |
| <q>  | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                    |
| <Sd> | Is the 32-bit name of the floating-point destination register, encoded in the "Vd:D" field. |
| <Dm> | Is the 64-bit name of the floating-point source register, encoded in the "M:Vm" field.      |
| <Dd> | Is the 64-bit name of the floating-point destination register, encoded in the "D:Vd" field. |
| <Sm> | Is the 32-bit name of the floating-point source register, encoded in the "Vm:M" field.      |

### Operation for all encodings

```

1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 ExecuteFPCheck();
4
5 if convert_from_half then
6 if dp_operation then
7 D[d] = FPHalfToDouble(S[m][lowbit+15:lowbit], TRUE);
8 else
9 S[d] = FPHalfToSingle(S[m][lowbit+15:lowbit], TRUE);
10 else
11 if dp_operation then
12 S[d][lowbit+15:lowbit] = FPDoubleToHalf(D[m], TRUE);
13 else
14 S[d][lowbit+15:lowbit] = FPSingleToHalf(S[m], TRUE);

```

## C2.4.261 VCVTM

Convert floating-point to integer with Round towards -Infinity. Convert floating-point to integer with Round towards -Infinity converts a value in a register from floating-point to a 32-bit integer using the Round towards -Infinity rounding mode, and places the result in the destination register.

### T1

*Armv8-M Floating-point Extension only*, *sz == 1* UNDEFINED in single-precision only implementations.

|    |    |    |    |    |    |   |   |   |   |   |   |   |   |         |    |    |    |    |    |    |    |   |   |    |   |   |   |   |   |   |   |
|----|----|----|----|----|----|---|---|---|---|---|---|---|---|---------|----|----|----|----|----|----|----|---|---|----|---|---|---|---|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1       | 0  | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7  | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 1  | 1  | 1  | 1  | 1  | 1  | 1 | 0 | 1 | D | 1 | 1 | 1 | 1 | RM = 11 | Vd | 1  | 0  | 1  | sz | op | 1  | M | 0 | Vm |   |   |   |   |   |   |   |

### Single-precision scalar variant

*Armv8-M Floating-point Extension only*.

Applies when **sz == 0**.

VCVTM{<q>}.<dt>.F32 <Sd>, <Sm>

### Double-precision scalar variant

*Armv8-M Floating-point Extension only*.

Applies when **sz == 1**.

VCVTM{<q>}.<dt>.F64 <Sd>, <Dm>

### Decode for this encoding

```

1 dp_operation = (sz == '1');
2 CheckDecodeFaults(dp_operation);
3 if InITBlock() then UNPREDICTABLE;
4 unsigned = (op == '0');
5 round_mode = RM;
6 d = UInt(Vd:D);
7 m = if dp_operation then UInt(M:Vm) else UInt(Vm:M);

```

### Assembler symbols for all encodings

<q> See [C1.2.5 Standard assembler syntax fields](#) on page 309.

<dt> Is the data type for the elements of the destination, encoded in the "op" field. It can have the following values:  
 U32 when op = 0  
 S32 when op = 1

<Sd> Is the 32-bit name of the floating-point destination register, encoded in the "Vd:D" field.

<Sm> Is the 32-bit name of the floating-point source register, encoded in the "Vm:M" field.

<Dm> Is the 64-bit name of the floating-point source register, encoded in the "M:Vm" field.

### Operation for all encodings

```

1 EncodingSpecificOperations();
2 ExecuteFPCheck();
3
4 if dp_operation then
5 S[d] = FPToFixedDirected(D[m], 0, unsigned, round_mode, TRUE);
6 else
7 S[d] = FPToFixedDirected(S[m], 0, unsigned, round_mode, TRUE);

```



## C2.4.262 VCVTN

Convert floating-point to integer with Round to Nearest. Convert floating-point to integer with Round to Nearest converts a value in a register from floating-point to a 32-bit integer using the Round to Nearest rounding mode, and places the result in the destination register.

### T1

*Armv8-M Floating-point Extension only*,  $sz == 1$  UNDEFINED in single-precision only implementations.

|    |    |    |    |    |    |   |   |   |   |   |   |   |   |        |    |    |    |    |    |    |    |   |   |    |   |   |   |   |   |   |   |
|----|----|----|----|----|----|---|---|---|---|---|---|---|---|--------|----|----|----|----|----|----|----|---|---|----|---|---|---|---|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1      | 0  | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7  | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 1  | 1  | 1  | 1  | 1  | 1  | 1 | 0 | 1 | D | 1 | 1 | 1 | 1 | RM = 0 | Vd | 1  | 0  | 1  | sz | op | 1  | M | 0 | Vm |   |   |   |   |   |   |   |

### Single-precision scalar variant

*Armv8-M Floating-point Extension only*.

Applies when  $sz == 0$ .

`VCVTN{<q>}.<dt>.F32 <Sd>, <Sm>`

### Double-precision scalar variant

*Armv8-M Floating-point Extension only*.

Applies when  $sz == 1$ .

`VCVTN{<q>}.<dt>.F64 <Sd>, <Dm>`

### Decode for this encoding

```

1 dp_operation = (sz == '1');
2 CheckDecodeFaults(dp_operation);
3 if InITBlock() then UNPREDICTABLE;
4 unsigned = (op == '0');
5 round_mode = RM;
6 d = UInt(Vd:D);
7 m = if dp_operation then UInt(M:Vm) else UInt(Vm:M);

```

### Assembler symbols for all encodings

|      |                                                                                                                                                          |
|------|----------------------------------------------------------------------------------------------------------------------------------------------------------|
| <q>  | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                                                                                 |
| <dt> | Is the data type for the elements of the destination, encoded in the "op" field. It can have the following values:<br>U32 when op = 0<br>S32 when op = 1 |
| <Sd> | Is the 32-bit name of the floating-point destination register, encoded in the "Vd:D" field.                                                              |
| <Sm> | Is the 32-bit name of the floating-point source register, encoded in the "Vm:M" field.                                                                   |
| <Dm> | Is the 64-bit name of the floating-point source register, encoded in the "M:Vm" field.                                                                   |

### Operation for all encodings

```

1 EncodingSpecificOperations();
2 ExecuteFPCheck();
3
4 if dp_operation then
5 S[d] = FPToFixedDirected(D[m], 0, unsigned, round_mode, TRUE);
6 else
7 S[d] = FPToFixedDirected(S[m], 0, unsigned, round_mode, TRUE);

```

### C2.4.263 VCVTP

Convert floating-point to integer with Round towards +Infinity. Convert floating-point to integer with Round towards +Infinity converts a value in a register from floating-point to a 32-bit integer using the Round towards +Infinity rounding mode, and places the result in the destination register.

#### T1

*Armv8-M Floating-point Extension only*, *sz == 1* UNDEFINED in single-precision only implementations.

|    |    |    |    |    |    |   |   |   |   |   |   |   |   |         |   |    |    |    |    |    |    |   |    |    |   |   |   |   |   |    |   |
|----|----|----|----|----|----|---|---|---|---|---|---|---|---|---------|---|----|----|----|----|----|----|---|----|----|---|---|---|---|---|----|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1       | 0 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8  | 7  | 6 | 5 | 4 | 3 | 2 | 1  | 0 |
| 1  | 1  | 1  | 1  | 1  | 1  | 1 | 0 | 1 | D | 1 | 1 | 1 | 1 | RM = 10 |   | Vd |    |    |    | 1  | 0  | 1 | sz | op | 1 | M | 0 |   |   | Vm |   |

#### Single-precision scalar variant

*Armv8-M Floating-point Extension only*.

Applies when **sz == 0**.

VCVTP{<q>}.<dt>.F32 <Sd>, <Sm>

#### Double-precision scalar variant

*Armv8-M Floating-point Extension only*.

Applies when **sz == 1**.

VCVTP{<q>}.<dt>.F64 <Sd>, <Dm>

#### Decode for this encoding

```

1 dp_operation = (sz == '1');
2 CheckDecodeFaults(dp_operation);
3 if InITBlock() then UNPREDICTABLE;
4 unsigned = (op == '0');
5 round_mode = RM;
6 d = UInt(Vd:D);
7 m = if dp_operation then UInt(M:Vm) else UInt(Vm:M);

```

#### Assembler symbols for all encodings

<q> See [C1.2.5 Standard assembler syntax fields](#) on page 309.

<dt> Is the data type for the elements of the destination, encoded in the "op" field. It can have the following values:  
 U32 when op = 0  
 S32 when op = 1

<Sd> Is the 32-bit name of the floating-point destination register, encoded in the "Vd:D" field.

<Sm> Is the 32-bit name of the floating-point source register, encoded in the "Vm:M" field.

<Dm> Is the 64-bit name of the floating-point source register, encoded in the "M:Vm" field.

#### Operation for all encodings

```

1 EncodingSpecificOperations();
2 ExecuteFPCheck();
3
4 if dp_operation then
5 S[d] = FPToFixedDirected(D[m], 0, unsigned, round_mode, TRUE);
6 else
7 S[d] = FPToFixedDirected(S[m], 0, unsigned, round_mode, TRUE);

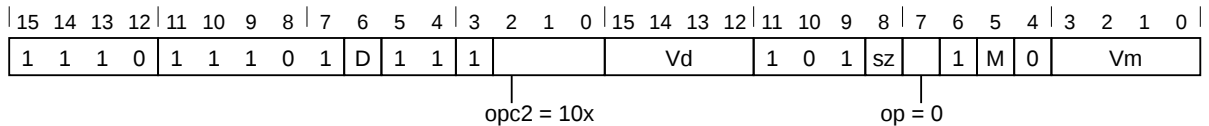
```

## C2.4.264 VCVTR

Convert floating-point to integer. Convert floating-point to integer converts a value in a register from floating-point to a 32-bit integer, using the rounding mode specified by FPSCR, and places the result in the destination register.

### T1

*Armv8-M Floating-point Extension only, sz == 1 UNDEFINED in single-precision only implementations.*



#### Single-precision scalar variant

*Armv8-M Floating-point Extension only.*

Applies when `opc2 == 100` && `sz == 0`.

`VCVTR{<c>}{<q>}.U32.F32 <Sd>, <Sm>`

#### Single-precision scalar variant

*Armv8-M Floating-point Extension only.*

Applies when `opc2 == 101` && `sz == 0`.

`VCVTR{<c>}{<q>}.S32.F32 <Sd>, <Sm>`

#### Double-precision scalar variant

*Armv8-M Floating-point Extension only*

Applies when `opc2 == 100` && `sz == 1`.

`VCVTR{<c>}{<q>}.U32.F64 <Sd>, <Dm>`

#### Double-precision scalar variant

*Armv8-M Floating-point Extension only*

Applies when `opc2 == 101` && `sz == 1`.

`VCVTR{<c>}{<q>}.S32.F64 <Sd>, <Dm>`

### Decode for this encoding

```

1 if opc2 != '000' && !(opc2 IN '10x') then SEE "Related encodings";
2 dp_operation = (sz == '1');
3 CheckDecodeFaults(dp_operation);
4 to_integer = (opc2[2] == '1');
5 if to_integer then
6 unsigned = (opc2[0] == '0'); round_zero = (op == '1');
7 d = UInt(Vd:D); m = if dp_operation then UInt(M:Vm) else UInt(Vm:M);
8 else
9 unsigned = (op == '0'); round_nearest = FALSE; // FALSE selects FPSCR rounding
10 m = UInt(Vm:M); d = if dp_operation then UInt(D:Vd) else UInt(Vd:D);

```

### Assembler symbols for all encodings

`<c>` See [C1.2.5 Standard assembler syntax fields](#) on page 309.  
`<q>` See [C1.2.5 Standard assembler syntax fields](#) on page 309.

|      |                                                                                             |
|------|---------------------------------------------------------------------------------------------|
| <Sd> | Is the 32-bit name of the floating-point destination register, encoded in the "Vd:D" field. |
| <Sm> | Is the 32-bit name of the floating-point source register, encoded in the "Vm:M" field.      |
| <Dm> | Is the 64-bit name of the floating-point source register, encoded in the "M:Vm" field.      |

### Operation for all encodings

```

1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 ExecuteFPCheck();
4 if to_integer then
5 if dp_operation then
6 S[d] = FPToFixed(D[m], 32, 0, unsigned, round_zero, TRUE);
7 else
8 S[d] = FPToFixed(S[m], 32, 0, unsigned, round_zero, TRUE);
9 else
10 if dp_operation then
11 D[d] = FixedToFP(S[m], 64, 0, unsigned, round_nearest, TRUE);
12 else
13 S[d] = FixedToFP(S[m], 32, 0, unsigned, round_nearest, TRUE);

```

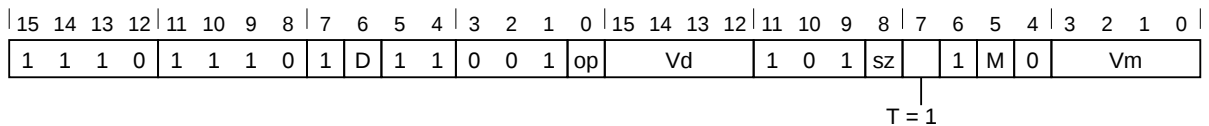
## C2.4.265 VCVTT

Floating-point Convert Top. Floating-point Convert Top does one of the following:

- Converts the half-precision value in the top half of a single-precision register to single-precision and writes the result to a single-precision register.
- Converts the value in a single-precision register to half-precision and writes the result into the top half of a single-precision register, preserving the other half of the target register.
- Converts the half-precision value in the top half of a single-precision register to double-precision and writes the result to a double-precision register, without intermediate rounding.
- Converts the value in the double-precision register to half-precision and writes the result into the top half of a double-precision register, preserving the other half of the target register, without intermediate rounding.

### T1

*Armv8-M Floating-point Extension only*,  $sz == 1$  UNDEFINED in single-precision only implementations.



#### Single-precision scalar variant

*Armv8-M Floating-point Extension only.*

Applies when  $op == 0$  &&  $sz == 0$ .

`VCVTT{<c>}{<q>}.F32.F16 <Sd>, <Sm>`

#### Single-precision scalar variant

*Armv8-M Floating-point Extension only.*

Applies when  $op == 1$  &&  $sz == 0$ .

`VCVTT{<c>}{<q>}.F16.F32 <Sd>, <Sm>`

#### Double-precision scalar variant

*Armv8-M Floating-point Extension only.*

Applies when  $op == 0$  &&  $sz == 1$ .

`VCVTT{<c>}{<q>}.F64.F16 <Dd>, <Sm>`

#### Double-precision scalar variant

*Armv8-M Floating-point Extension only.*

Applies when  $op == 1$  &&  $sz == 1$ .

`VCVTT{<c>}{<q>}.F16.F64 <Sd>, <Dm>`

### Decode for this encoding

```

1 dp_operation = (sz == '1');
2 CheckDecodeFaults(dp_operation);
3 convert_from_half = (op == '0');
4 lowbit = if T == '1' then 16 else 0;
5 if dp_operation then

```

```

6 if convert_from_half then
7 d = UInt(D:Vd); m = UInt(Vm:M);
8 else
9 d = UInt(Vd:D); m = UInt(M:Vm);
10 else
11 d = UInt(Vd:D); m = UInt(Vm:M);

```

### Assembler symbols for all encodings

|      |                                                                                             |
|------|---------------------------------------------------------------------------------------------|
| <c>  | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                    |
| <q>  | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                    |
| <Sd> | Is the 32-bit name of the floating-point destination register, encoded in the "Vd:D" field. |
| <Dm> | Is the 64-bit name of the floating-point source register, encoded in the "M:Vm" field.      |
| <Dd> | Is the 64-bit name of the floating-point destination register, encoded in the "D:Vd" field. |
| <Sm> | Is the 32-bit name of the floating-point source register, encoded in the "Vm:M" field.      |

### Operation for all encodings

```

1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 ExecuteFPCheck();
4
5 if convert_from_half then
6 if dp_operation then
7 D[d] = FPHalfToDouble(S[m][lowbit+15:lowbit], TRUE);
8 else
9 S[d] = FPHalfToSingle(S[m][lowbit+15:lowbit], TRUE);
10 else
11 if dp_operation then
12 S[d][lowbit+15:lowbit] = FPDoubleToHalf(D[m], TRUE);
13 else
14 S[d][lowbit+15:lowbit] = FPSingleToHalf(S[m], TRUE);

```

## C2.4.266 VDIV

Floating-point Divide. Floating-point Divide divides one floating-point value by another floating-point value and writes the result to a third floating-point register.

### T1

*Armv8-M Floating-point Extension only*, *sz == 1* UNDEFINED in single-precision only implementations.

|    |    |    |    |    |    |   |   |   |   |   |   |    |   |   |   |    |    |    |    |    |    |   |    |   |   |   |   |    |   |   |   |
|----|----|----|----|----|----|---|---|---|---|---|---|----|---|---|---|----|----|----|----|----|----|---|----|---|---|---|---|----|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3  | 2 | 1 | 0 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8  | 7 | 6 | 5 | 4 | 3  | 2 | 1 | 0 |
| 1  | 1  | 1  | 0  | 1  | 1  | 1 | 0 | 1 | D | 0 | 0 | Vn |   |   |   | Vd |    |    |    | 1  | 0  | 1 | sz | N | 0 | M | 0 | Vm |   |   |   |

#### Single-precision scalar variant

*Armv8-M Floating-point Extension only*.

Applies when *sz == 0*.

VDIV{<c>}{<q>}.F32 {<Sd>}, {<Sn>}, {<Sm>}

#### Double-precision scalar variant

*Armv8-M Floating-point Extension only*.

Applies when *sz == 1*.

VDIV{<c>}{<q>}.F64 {<Dd>}, {<Dn>}, {<Dm>}

#### Decode for this encoding

```

1 dp_operation = (sz == '1');
2 CheckDecodeFaults(dp_operation);
3 d = if dp_operation then UInt(D:Vd) else UInt(Vd:D);
4 n = if dp_operation then UInt(N:Vn) else UInt(Vn:N);
5 m = if dp_operation then UInt(M:Vm) else UInt(Vm:M);

```

#### Assembler symbols for all encodings

|      |                                                                                               |
|------|-----------------------------------------------------------------------------------------------|
| <c>  | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                      |
| <q>  | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                      |
| <Sd> | Is the 32-bit name of the floating-point destination register, encoded in the "Vd:D" field.   |
| <Sn> | Is the 32-bit name of the first floating-point source register, encoded in the "Vn:N" field.  |
| <Sm> | Is the 32-bit name of the second floating-point source register, encoded in the "Vm:M" field. |
| <Dd> | Is the 64-bit name of the floating-point destination register, encoded in the "D:Vd" field.   |
| <Dn> | Is the 64-bit name of the first floating-point source register, encoded in the "N:Vn" field.  |
| <Dm> | Is the 64-bit name of the second floating-point source register, encoded in the "M:Vm" field. |

#### Operation for all encodings

```

1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 ExecuteFPCheck();
4 if dp_operation then
5 D[d] = FPDiv(D[n], D[m], TRUE);
6 else
7 S[d] = FPDiv(S[n], S[m], TRUE);

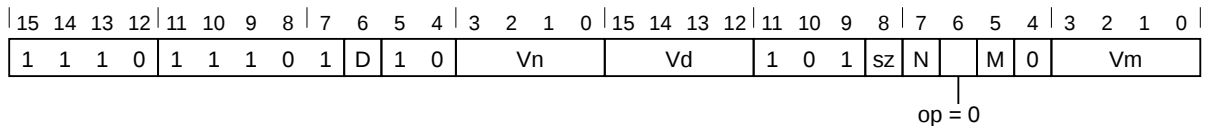
```

## C2.4.267 VFMA

Floating-point Fused Multiply Accumulate. Floating-point Fused Multiply Accumulate multiplies two registers, adds the product to the destination register, and places the result in the destination register. The result of the multiply is not rounded before the addition.

### T2

Armv8-M Floating-point Extension only, *sz* == 1 UNDEFINED in single-precision only implementations.



#### Single-precision scalar variant

Armv8-M Floating-point Extension only.

Applies when *sz* == 0.

VFMA{<c>}{<q>}.F32 <Sd>, <Sn>, <Sm>

#### Double-precision scalar variant

Armv8-M Floating-point Extension only.

Applies when *sz* == 1.

VFMA{<c>}{<q>}.F64 <Dd>, <Dn>, <Dm>

#### Decode for this encoding

```

1 dp_operation = (sz == '1');
2 CheckDecodeFaults(dp_operation);
3 opl_neg = (op == '1');
4 d = if dp_operation then UInt(D:Vd) else UInt(Vd:D);
5 n = if dp_operation then UInt(N:Vn) else UInt(Vn:N);
6 m = if dp_operation then UInt(M:Vm) else UInt(Vm:M);

```

#### Assembler symbols for all encodings

|                         |                                                                                               |
|-------------------------|-----------------------------------------------------------------------------------------------|
| <code>&lt;c&gt;</code>  | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                      |
| <code>&lt;q&gt;</code>  | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                      |
| <code>&lt;Sd&gt;</code> | Is the 32-bit name of the floating-point destination register, encoded in the "Vd:D" field.   |
| <code>&lt;Sn&gt;</code> | Is the 32-bit name of the first floating-point source register, encoded in the "Vn:N" field.  |
| <code>&lt;Sm&gt;</code> | Is the 32-bit name of the second floating-point source register, encoded in the "Vm:M" field. |
| <code>&lt;Dd&gt;</code> | Is the 64-bit name of the floating-point destination register, encoded in the "D:Vd" field.   |
| <code>&lt;Dn&gt;</code> | Is the 64-bit name of the first floating-point source register, encoded in the "N:Vn" field.  |
| <code>&lt;Dm&gt;</code> | Is the 64-bit name of the second floating-point source register, encoded in the "M:Vm" field. |

#### Operation for all encodings

```

1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 ExecuteFPCheck();
4 if dp_operation then
5 op64 = if opl_neg then FPNeg(D[n]) else D[n];
6 D[d] = FPMulAdd(D[d], op64, D[m], TRUE);
7 else
8 op32 = if opl_neg then FPNeg(S[n]) else S[n];
9 S[d] = FPMulAdd(S[d], op32, S[m], TRUE);

```

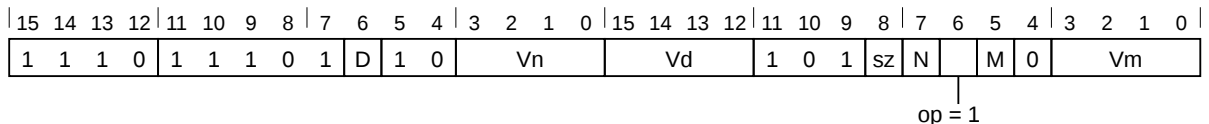


## C2.4.268 VFMS

Floating-point Fused Multiply Subtract. Floating-point Fused Multiply Subtract negates one register and multiplies it with another register, adds the product to the destination register, and places the result in the destination register. The result of the multiply is not rounded before the addition.

### T2

Armv8-M Floating-point Extension only, `sz == 1` UNDEFINED in single-precision only implementations.



#### Single-precision scalar variant

Armv8-M Floating-point Extension only.

Applies when `sz == 0`.

VFMS{<c>}{<q>}.F32 <Sd>, <Sn>, <Sm>

#### Double-precision scalar variant

Armv8-M Floating-point Extension only.

Applies when `sz == 1`.

VFMS{<c>}{<q>}.F64 <Dd>, <Dn>, <Dm>

#### Decode for this encoding

```

1 dp_operation = (sz == '1');
2 CheckDecodeFaults(dp_operation);
3 opl_neg = (op == '1');
4 d = if dp_operation then UInt(D:Vd) else UInt(Vd:D);
5 n = if dp_operation then UInt(N:Vn) else UInt(Vn:N);
6 m = if dp_operation then UInt(M:Vm) else UInt(Vm:M);

```

#### Assembler symbols for all encodings

|                         |                                                                                               |
|-------------------------|-----------------------------------------------------------------------------------------------|
| <code>&lt;c&gt;</code>  | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                      |
| <code>&lt;q&gt;</code>  | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                      |
| <code>&lt;Sd&gt;</code> | Is the 32-bit name of the floating-point destination register, encoded in the "Vd:D" field.   |
| <code>&lt;Sn&gt;</code> | Is the 32-bit name of the first floating-point source register, encoded in the "Vn:N" field.  |
| <code>&lt;Sm&gt;</code> | Is the 32-bit name of the second floating-point source register, encoded in the "Vm:M" field. |
| <code>&lt;Dd&gt;</code> | Is the 64-bit name of the floating-point destination register, encoded in the "D:Vd" field.   |
| <code>&lt;Dn&gt;</code> | Is the 64-bit name of the first floating-point source register, encoded in the "N:Vn" field.  |
| <code>&lt;Dm&gt;</code> | Is the 64-bit name of the second floating-point source register, encoded in the "M:Vm" field. |

#### Operation for all encodings

```

1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 ExecuteFPCheck();
4 if dp_operation then
5 op64 = if opl_neg then FPNeg(D[n]) else D[n];
6 D[d] = FPMulAdd(D[d], op64, D[m], TRUE);
7 else
8 op32 = if opl_neg then FPNeg(S[n]) else S[n];
9 S[d] = FPMulAdd(S[d], op32, S[m], TRUE);

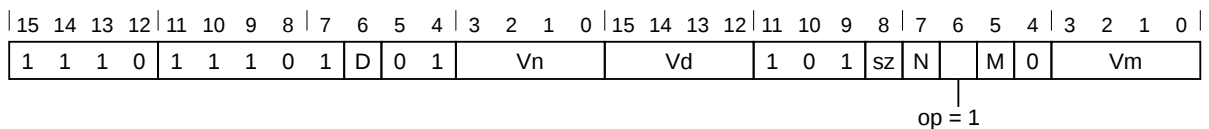
```

## C2.4.269 VFNMA

Floating-point Fused Negate Multiply Accumulate. Floating-point Fused Negate Multiply Accumulate negates one floating-point register value and multiplies it by another floating-point register value, adds the negation of the floating-point value in the destination register to the product, and writes the result back to the destination register. The result of the multiply is not rounded before the addition.

### T1

Armv8-M Floating-point Extension only, *sz* == 1 UNDEFINED in single-precision only implementations.



#### Single-precision scalar variant

Armv8-M Floating-point Extension only.

Applies when *sz* == 0.

VFNMA{<c>}{<q>}.F32 <Sd>, <Sn>, <Sm>

#### Double-precision scalar variant

Armv8-M Floating-point Extension only.

Applies when *sz* == 1.

VFNMA{<c>}{<q>}.F64 <Dd>, <Dn>, <Dm>

#### Decode for this encoding

```

1 dp_operation = (sz == '1');
2 CheckDecodeFaults(dp_operation);
3 opl_neg = (op == '1');
4 d = if dp_operation then UInt(D:Vd) else UInt(Vd:D);
5 n = if dp_operation then UInt(N:Vn) else UInt(Vn:N);
6 m = if dp_operation then UInt(M:Vm) else UInt(Vm:M);

```

#### Assembler symbols for all encodings

|                         |                                                                                               |
|-------------------------|-----------------------------------------------------------------------------------------------|
| <code>&lt;c&gt;</code>  | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                      |
| <code>&lt;q&gt;</code>  | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                      |
| <code>&lt;Sd&gt;</code> | Is the 32-bit name of the floating-point destination register, encoded in the "Vd:D" field.   |
| <code>&lt;Sn&gt;</code> | Is the 32-bit name of the first floating-point source register, encoded in the "Vn:N" field.  |
| <code>&lt;Sm&gt;</code> | Is the 32-bit name of the second floating-point source register, encoded in the "Vm:M" field. |
| <code>&lt;Dd&gt;</code> | Is the 64-bit name of the floating-point destination register, encoded in the "D:Vd" field.   |
| <code>&lt;Dn&gt;</code> | Is the 64-bit name of the first floating-point source register, encoded in the "N:Vn" field.  |
| <code>&lt;Dm&gt;</code> | Is the 64-bit name of the second floating-point source register, encoded in the "M:Vm" field. |

#### Operation for all encodings

```

1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 ExecuteFPCheck();
4 if dp_operation then
5 op64 = if opl_neg then FPNeg(D[n]) else D[n];
6 D[d] = FPMulAdd(FPNeg(D[d]), op64, D[m], TRUE);
7 else

```

## Chapter C2. Instruction Specification

### C2.4. Alphabetical list of instructions

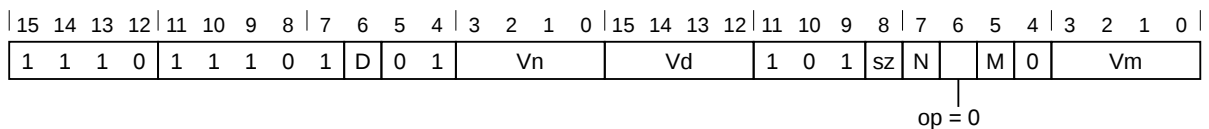
```
8 op32 = if op1_neg then FPNeg(S[n]) else S[n];
9 S[d] = FPMulAdd(FPNeg(S[d]), op32, S[m], TRUE);
```

## C2.4.270 VFNMS

Floating-point Fused Negate Multiply Subtract. Floating-point Fused Negate Multiply Subtract multiplies together two floating-point register values, adds the negation of the floating-point value in the destination register to the product, and writes the result back to the destination register. The result of the multiply is not rounded before the addition.

### T1

Armv8-M Floating-point Extension only, `sz == 1` UNDEFINED in single-precision only implementations.



#### Single-precision scalar variant

Armv8-M Floating-point Extension only.

Applies when `sz == 0`.

VFNMS {<c>} {<q>} .F32 <Sd>, <Sn>, <Sm>

#### Double-precision scalar variant

Armv8-M Floating-point Extension only.

Applies when `sz == 1`.

VFNMS {<c>} {<q>} .F64 <Dd>, <Dn>, <Dm>

#### Decode for this encoding

```

1 dp_operation = (sz == '1');
2 CheckDecodeFaults(dp_operation);
3 opl_neg = (op == '1');
4 d = if dp_operation then UInt(D:Vd) else UInt(Vd:D);
5 n = if dp_operation then UInt(N:Vn) else UInt(Vn:N);
6 m = if dp_operation then UInt(M:Vm) else UInt(Vm:M);

```

#### Assembler symbols for all encodings

|                         |                                                                                               |
|-------------------------|-----------------------------------------------------------------------------------------------|
| <code>&lt;c&gt;</code>  | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                      |
| <code>&lt;q&gt;</code>  | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                      |
| <code>&lt;Sd&gt;</code> | Is the 32-bit name of the floating-point destination register, encoded in the "Vd:D" field.   |
| <code>&lt;Sn&gt;</code> | Is the 32-bit name of the first floating-point source register, encoded in the "Vn:N" field.  |
| <code>&lt;Sm&gt;</code> | Is the 32-bit name of the second floating-point source register, encoded in the "Vm:M" field. |
| <code>&lt;Dd&gt;</code> | Is the 64-bit name of the floating-point destination register, encoded in the "D:Vd" field.   |
| <code>&lt;Dn&gt;</code> | Is the 64-bit name of the first floating-point source register, encoded in the "N:Vn" field.  |
| <code>&lt;Dm&gt;</code> | Is the 64-bit name of the second floating-point source register, encoded in the "M:Vm" field. |

#### Operation for all encodings

```

1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 ExecuteFPCheck();
4 if dp_operation then
5 op64 = if opl_neg then FPNeg(D[n]) else D[n];
6 D[d] = FPMulAdd(FPNeg(D[d]), op64, D[m], TRUE);
7 else

```

Chapter C2. Instruction Specification

C2.4. Alphabetical list of instructions

```
8 op32 = if op1_neg then FPNeg(S[n]) else S[n];
9 S[d] = FPMulAdd(FPNeg(S[d]), op32, S[m], TRUE);
```

### C2.4.271 VLDM

Floating-point Load Multiple. Floating-point Load Multiple loads multiple extension registers from consecutive memory locations using an address from a general-purpose register.

This instruction is used by the alias [VPOP](#).

#### T1

*Armv8-M Floating-point Extension only*

|    |    |    |    |    |    |   |   |   |   |   |   |    |   |   |   |    |    |    |    |    |    |   |   |             |   |   |   |   |   |   |   |
|----|----|----|----|----|----|---|---|---|---|---|---|----|---|---|---|----|----|----|----|----|----|---|---|-------------|---|---|---|---|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3  | 2 | 1 | 0 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7           | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 1  | 1  | 1  | 0  | 1  | 1  | 0 | P | U | D | W | 1 | Rn |   |   |   | Vd |    |    |    | 1  | 0  | 1 | 1 | imm8<0> = 0 |   |   |   |   |   |   |   |

#### Decrement Before variant

Applies when **P == 1 && U == 0 && W == 1**.

VLDMDB{<c>}{<q>}{.<size>} <Rn>!, <dreglist>

#### Increment After variant

Applies when **P == 0 && U == 1**.

VLDM{<c>}{<q>}{.<size>} <Rn>{!}, <dreglist>

VLDMIA{<c>}{<q>}{.<size>} <Rn>{!}, <dreglist>

#### Decode for this encoding

```

1 if P == '0' && U == '0' && W == '0' then SEE "Related encodings";
2 if P == '1' && W == '0' then SEE VLDR;
3 CheckDecodeFaults();
4 if P == U && W == '1' then UNDEFINED;
5 // Remaining combinations are PUW = 010 (IA without !), 011 (IA with !), 101 (DB with !)
6 single_regs = FALSE; add = (U == '1'); wback = (W == '1');
7 d = UInt(D:Vd); n = UInt(Rn); imm32 = ZeroExtend(imm8:'00', 32);
8 regs = UInt(imm8) DIV 2;
9 if n == 15 then UNPREDICTABLE;
10 if regs == 0 || regs > 16 || (d+regs) > 32 then UNPREDICTABLE;
11 if VFPSmallRegisterBank() && (d+regs) > 16 then UNPREDICTABLE;
```

#### CONSTRAINED UNPREDICTABLE behavior

If `regs == 0`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction operates as a [VLDM](#) with the same addressing mode but loads no registers.

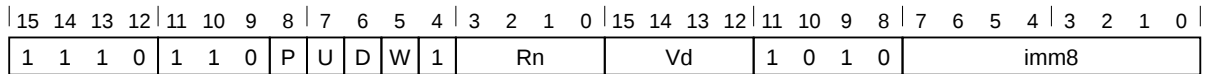
#### CONSTRAINED UNPREDICTABLE behavior

If `(d+regs) > 32`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- One or more of the floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

## T2

*Armv8-M Floating-point Extension only*



### Decrement Before variant

Applies when **P == 1 && U == 0 && W == 1**.

VLDMDB{<c>}{<q>}{.<size>} <Rn>!, <sreglist>

### Increment After variant

Applies when **P == 0 && U == 1**.

VLDM{<c>}{<q>}{.<size>} <Rn>{!}, <sreglist>

VLDMIA{<c>}{<q>}{.<size>} <Rn>{!}, <sreglist>

### Decode for this encoding

```

1 if P == '0' && U == '0' then SEE "Related encodings";
2 if P == '1' && W == '0' then SEE VLDR;
3 CheckDecodeFaults();
4 if P == '1' && U == '1' && W == '1' then UNDEFINED;
5 // Remaining combinations are P UW = 010 (IA without !), 011 (IA with !), 101 (DB with !)
6 single_regs = TRUE; add = (U == '1'); wback = (W == '1');
7 d = UInt(Vd:D); n = UInt(Rn); imm32 = ZeroExtend(imm8:'00', 32);
8 regs = UInt(imm8);
9 if n == 15 then UNPREDICTABLE;
10 if regs == 0 || (d+regs) > 32 then UNPREDICTABLE;
```

### CONSTRAINED UNPREDICTABLE behavior

If `regs == 0`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction operates as a **VLDM** with the same addressing mode but loads no registers.

### CONSTRAINED UNPREDICTABLE behavior

If `(d+regs) > 64`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- One or more of the floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

### Alias conditions

| Alias       | preferred when                                                                        |
|-------------|---------------------------------------------------------------------------------------|
| <b>VPOP</b> | <code>P == '0' &amp;&amp; U == '1' &amp;&amp; W == '1' &amp;&amp; Rn == '1101'</code> |

### Assembler symbols for all encodings

<c> See [C1.2.5 Standard assembler syntax fields](#) on page 309.  
 <q> See [C1.2.5 Standard assembler syntax fields](#) on page 309.

|            |                                                                                                                                                                                                                                                                                                         |
|------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <size>     | An optional data size specifier. If present, it must be equal to the size in bits, 32 or 64, of the registers being transferred.                                                                                                                                                                        |
| <Rn>       | Is the general-purpose base register, encoded in the "Rn" field.                                                                                                                                                                                                                                        |
| !          | Specifies base register writeback. Encoded in the "W" field as 1 if present, otherwise 0.                                                                                                                                                                                                               |
| <sreglist> | Is the list of consecutively numbered 32-bit floating-point registers to be transferred. The first register in the list is encoded in "Vd:D", and "imm8" is set to the number of registers in the list. The list must contain at least one register.                                                    |
| <dreglist> | Is the list of consecutively numbered 64-bit floating-point registers to be transferred. The first register in the list is encoded in "D:Vd", and "imm8" is set to twice the number of registers in the list. The list must contain at least one register, and must not contain more than 16 registers. |

### Operation for all encodings

```

1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 ExecuteFPCheck();
4 address = if add then R[n] else R[n]-imm32;
5 regval = if add then R[n]+imm32 else R[n]-imm32;
6
7 // Determine if the stack pointer limit must be checked
8 if n == 13 && wback then
9 (limit, applylimit) = LookUpSPLim(LookUpSP());
10 // If memory operation is not performed as a result of a stack limit violation,
11 // and the write-back of the SP itself does not raise a stack limit violation, it
12 // is "IMPLEMENTATION_DEFINED" whether a SPLIM exception is raised.
13 // Arm recommends that any instruction which discards a memory access as
14 // a result of a stack limit violation, and where the write-back of the SP itself
15 // does not raise a stack limit violation, generates an SPLIM exception.
16 if boolean IMPLEMENTATION_DEFINED "SPLIM exception on invalid memory access" then
17 if applylimit && (UInt(address) < UInt(limit)) then
18 if HaveMainExt() then
19 UFSR.STKOF = '1';
20 // If Main Extension is not implemented the fault always escalates to
21 // HardFault
22 excInfo = CreateException(UsageFault, FALSE, boolean UNKNOWN);
23 HandleException(excInfo);
24 else
25 applylimit = FALSE;
26
27 // Memory operation only performed if limit not violated
28 if !applylimit || (UInt(regval) >= UInt(limit)) then
29 for r = 0 to regs-1
30 if single_regs then
31 S[d+r] = MemA[address,4];
32 address = address+4;
33 else
34 word1 = MemA[address,4]; word2 = MemA[address+4,4];
35 address = address+8;
36 // Combine the word-aligned words in the correct order for
37 // current endianness.
38 D[d+r] = if BigEndian() then word1:word2 else word2:word1;
39
40 // If the stack pointer is being updated a fault will be raised if
41 // the limit is violated
42 if wback then RSPCheck[n] = regval;

```



## C2.4.272 VLDR

Floating-point Load Register. Floating-point Load Register loads a Floating-point Extension register from memory, using an address from a general-purpose register, with an optional offset.

### T1

*Armv8-M Floating-point Extension only*

|    |    |    |    |    |    |   |   |   |   |   |   |    |   |   |   |    |    |    |    |         |    |   |   |      |   |   |   |   |   |   |   |
|----|----|----|----|----|----|---|---|---|---|---|---|----|---|---|---|----|----|----|----|---------|----|---|---|------|---|---|---|---|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3  | 2 | 1 | 0 | 15 | 14 | 13 | 12 | 11      | 10 | 9 | 8 | 7    | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 1  | 1  | 1  | 0  | 1  | 1  | 0 | 1 | U | D | 0 | 1 | Rn |   |   |   | Vd |    |    |    | 1 0 1 1 |    |   |   | imm8 |   |   |   |   |   |   |   |

#### Literal variant

Applies when **Rn == 1111**.

```
VLDR{<c>}{<q>}{.64} <Dd>, <label>
VLDR{<c>}{<q>}{.64} <Dd>, [PC, #{+/-}<imm>]
```

#### Offset variant

Applies when **Rn != 1111**.

```
VLDR{<c>}{<q>}{.64} <Dd>, [<Rn> {, #{+/-}<imm>}]
```

#### Decode for this encoding

```
1 CheckDecodeFaults();
2 single_reg = FALSE; add = (U == '1'); imm32 = ZeroExtend(imm8:'00', 32);
3 d = UInt(D:Vd); n = UInt(Rn);
```

### T2

*Armv8-M Floating-point Extension only*

|    |    |    |    |    |    |   |   |   |   |   |   |    |   |   |   |    |    |    |    |         |    |   |   |      |   |   |   |   |   |   |   |
|----|----|----|----|----|----|---|---|---|---|---|---|----|---|---|---|----|----|----|----|---------|----|---|---|------|---|---|---|---|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3  | 2 | 1 | 0 | 15 | 14 | 13 | 12 | 11      | 10 | 9 | 8 | 7    | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 1  | 1  | 1  | 0  | 1  | 1  | 0 | 1 | U | D | 0 | 1 | Rn |   |   |   | Vd |    |    |    | 1 0 1 0 |    |   |   | imm8 |   |   |   |   |   |   |   |

#### Literal variant

Applies when **Rn == 1111**.

```
VLDR{<c>}{<q>}{.32} <Sd>, <label>
VLDR{<c>}{<q>}{.32} <Sd>, [PC, #{+/-}<imm>]
```

#### Offset variant

Applies when **Rn != 1111**.

```
VLDR{<c>}{<q>}{.32} <Sd>, [<Rn> {, #{+/-}<imm>}]
```

#### Decode for this encoding

```
1 CheckDecodeFaults();
2 single_reg = TRUE; add = (U == '1'); imm32 = ZeroExtend(imm8:'00', 32);
3 d = UInt(Vd:D); n = UInt(Rn);
```

#### Assembler symbols for all encodings

<c> See [C1.2.5 Standard assembler syntax fields](#) on page 309.

|         |                                                                                                                                                                                                                                                                                                                                                                                                                            |
|---------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <q>     | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                                                                                                                                                                                                                                                                                                                                                   |
| .64     | Optional data size specifiers.                                                                                                                                                                                                                                                                                                                                                                                             |
| <Dd>    | The destination register for a doubleword load.                                                                                                                                                                                                                                                                                                                                                                            |
| .32     | Optional data size specifiers.                                                                                                                                                                                                                                                                                                                                                                                             |
| <Sd>    | The destination register for a singleword load.                                                                                                                                                                                                                                                                                                                                                                            |
| <label> | The label of the literal data item to be loaded. The assembler calculates the required value of the offset from the <a href="#">Align</a> (PC, 4) value of the instruction to this label. Permitted values are multiples of 4 in the range -1020 to 1020. If the offset is zero or positive, imm32 is equal to the offset and add == TRUE. If the offset is negative, imm32 is equal to minus the offset and add == FALSE. |
| <Rn>    | Is the general-purpose base register, encoded in the "Rn" field.                                                                                                                                                                                                                                                                                                                                                           |
| +/-     | Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in the "U" field. It can have the following values: <ul style="list-style-type: none"><li>- when U = 0</li><li>+ when U = 1</li></ul>                                                                                                                                                                        |
| <imm>   | The immediate offset used for forming the address. For the immediate forms of the syntax, <imm> can be omitted, in which case the #0 form of the instruction is assembled. Permitted values are multiples of 4 in the range 0 to 1020.                                                                                                                                                                                     |

### Operation for all encodings

```
1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 ExecuteFPCheck();
4 base = if n == 15 then Align(PC,4) else R[n];
5 address = if add then (base + imm32) else (base - imm32);
6 if single_reg then
7 S[d] = MemA[address,4];
8 else
9 word1 = MemA[address,4]; word2 = MemA[address+4,4];
10 // Combine the word-aligned words in the correct order for current endianness.
11 D[d] = if BigEndian() then word1:word2 else word2:word1;
```

### C2.4.273 VLLDM

Floating-point Lazy Load Multiple. Floating-point Lazy Load Multiple restores the contents of the Secure floating-point registers that were protected by a VLSTM instruction, and marks the floating-point context as active.

If the lazy state preservation set up by a previous VLSTM instruction is active (FPCCR.LSPACT == 1), this instruction deactivates lazy state preservation and enables access to the Secure floating-point registers.

If lazy state preservation is inactive (FPCCR.LSPACT == 0), either because lazy state preservation was not enabled (FPCCR.LSPEN == 0) or because a floating-point instruction caused the Secure floating-point register contents to be stored to memory, this instruction loads the stored Secure floating-point register contents back into the floating-point registers.

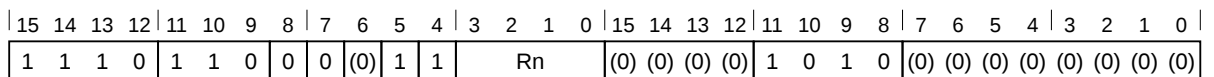
If Secure floating-point is not in use (CONTROL\_S.SFPA == 0), this instruction behaves as a NOP.

This instruction is only available in Secure state, and is UNDEFINED in Non-secure state.

If the Floating-point Extension is not implemented, this instruction is available in Secure state, but behaves as a NOP.

#### T1

Armv8-M Main Extension only



#### T1 variant

VLLDM{<c>} {<q>} <Rn>

#### Decode for this encoding

```

1 if !HaveMainExt() then UNDEFINED;
2 n = UInt(Rn);
3 if !IsSecure() then UNDEFINED;
4 if n == 15 then UNPREDICTABLE;

```

#### Assembler symbols for all encodings

- <c> See [C1.2.5 Standard assembler syntax fields](#) on page 309.
- <q> See [C1.2.5 Standard assembler syntax fields](#) on page 309.
- <Rn> Is the general-purpose base register, encoded in the "Rn" field.

#### Operation for all encodings

```

1 if ConditionPassed() then
2 EncodingSpecificOperations();
3
4 if CONTROL_S.SFPA == '1' then
5 // Check access to the co-processor is permitted
6 exc = CheckCPEnabled(10);
7 HandleException(exc);
8
9 if FPCCR_S.LSPACT == '1' then // state in FP is still valid
10 FPCCR_S.LSPACT = '0';
11 else
12 if !IsAligned(R[n], 8) then
13 UFSR.UNALIGNED = '1';
14 exc = CreateException(UsageFault, FALSE, boolean UNKNOWN);
15 HandleException(exc);
16

```

```
17 for i = 0 to 15
18 S[i] = MemA[R[n] + (4*i), 4];
19 FPSCR = MemA[R[n] + 0x40, 4];
20 if FPCCR_S.TS == '1' then
21 for i = 0 to 15
22 S[i+16] = MemA[R[n] + 0x48 + (4*i), 4];
23 CONTROL.FPCA = '1';
```

## C2.4.274 VLSTM

Floating-point Lazy Store Multiple. Floating-point Lazy Store Multiple stores the contents of Secure floating-point registers to a prepared stack frame, and clears the Secure floating-point registers.

If floating-point lazy preservation is enabled (FPCCR.LSPEN == 1), then the next time a floating-point instruction other than VLSTM or VLLDM is executed:

- The contents of Secure floating-point registers are stored to memory.
- The Secure floating-point registers are cleared.

If Secure floating-point is not in use (CONTROL\_S.SFPA == 0), this instruction behaves as a NOP.

This instruction is only available in Secure state, and is UNDEFINED in Non-secure state.

If the Floating-point Extension is not implemented, this instruction is available in Secure state, but behaves as a NOP.

### T1

Armv8-M Main Extension only

|    |    |    |    |    |    |   |   |   |     |   |   |    |   |   |   |     |     |     |     |    |    |   |   |     |     |     |     |     |     |     |     |
|----|----|----|----|----|----|---|---|---|-----|---|---|----|---|---|---|-----|-----|-----|-----|----|----|---|---|-----|-----|-----|-----|-----|-----|-----|-----|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6   | 5 | 4 | 3  | 2 | 1 | 0 | 15  | 14  | 13  | 12  | 11 | 10 | 9 | 8 | 7   | 6   | 5   | 4   | 3   | 2   | 1   | 0   |
| 1  | 1  | 1  | 0  | 1  | 1  | 0 | 0 | 0 | (0) | 1 | 0 | Rn |   |   |   | (0) | (0) | (0) | (0) | 1  | 0  | 1 | 0 | (0) | (0) | (0) | (0) | (0) | (0) | (0) | (0) |

### T1 variant

VLSTM{<c>} {<q>} <Rn>

### Decode for this encoding

```

1 if !HaveMainExt() then UNDEFINED;
2 n = UInt(Rn);
3 if !IsSecure() then UNDEFINED;
4 if n == 15 then UNPREDICTABLE;
```

### Assembler symbols for all encodings

<c> See C1.2.5 *Standard assembler syntax fields* on page 309.  
 <q> See C1.2.5 *Standard assembler syntax fields* on page 309.  
 <Rn> Is the general-purpose base register, encoded in the "Rn" field.

### Operation for all encodings

```

1 if ConditionPassed() then
2 EncodingSpecificOperations();
3
4 if CONTROL_S.SFPA == '1' then
5 // Check access to the co-processor is permitted
6 exc = CheckCPEnabled(10);
7 HandleException(exc);
8
9 // LSPACT should not be active at the same time as there is active FP
10 // state. This is a possible attack scenario so raise a SecureFault.
11 lspact = if FPCCR_S.S == '1' then FPCCR_S.LSPACT else FPCCR_NS.LSPACT;
12 if lspact == '1' then
13 SFSR.LSERR = '1';
14 exc = CreateException(SecureFault, TRUE, TRUE);
15 HandleException(exc);
16 else
17 if !IsAligned(R[n], 8) then
18 UFSR.UNALIGNED = '1';
```

```
19 exc = CreateException(UsageFault, FALSE, boolean UNKNOWN);
20 HandleException(exc);
21
22 if FPCCR.LSPEN == '0' then
23 for i = 0 to 15
24 MemA[R[n] + (4*i), 4] = S[i];
25 MemA[R[n] + 0x40, 4] = FPSCR;
26 if FPCCR.TS == '1' then
27 for i = 0 to 15
28 MemA[R[n] + 0x48 + (4*i), 4] = S[i+16];
29 S[i+16] = Zeros(32);
30 S[i] = Zeros(32);
31 FPSCR = Zeros(32);
32 else
33 for i = 0 to 15
34 S[i] = bits(32) UNKNOWN;
35 FPSCR = bits(32) UNKNOWN;
36 else
37 UpdateFPCCR(R[n], FALSE);
38 CONTROL.FPCA = '0';
```

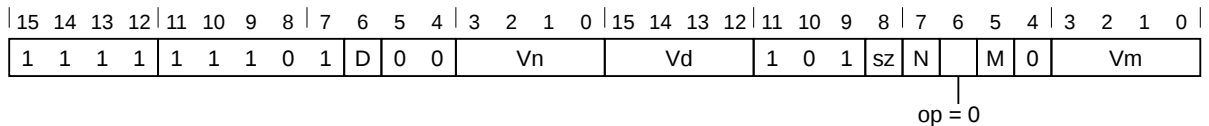
## C2.4.275 VMAXNM

Floating-point Maximum Number. Floating-point Maximum Number determines the floating-point maximum number.

NaN handling is specified by IEEE754-2008.

### T2

*Armv8-M Floating-point Extension only*, *sz == 1* UNDEFINED in single-precision only implementations.



#### Single-precision scalar variant

*Armv8-M Floating-point Extension only*.

Applies when **sz == 0**.

```
VMAXNM{<q>}.F32 <Sd>, <Sn>, <Sm>
 // Not permitted in IT block
```

#### Double-precision scalar variant

*Armv8-M Floating-point Extension only*.

Applies when **sz == 1**.

```
VMAXNM{<q>}.F64 <Dd>, <Dn>, <Dm>
 // Not permitted in IT block
```

#### Decode for this encoding

```
1 dp_operation = (sz == '1');
2 CheckDecodeFaults(dp_operation);
3 if InITBlock() then UNPREDICTABLE;
4 maximum = (op == '0');
5 d = if dp_operation then UInt(D:Vd) else UInt(Vd:D);
6 n = if dp_operation then UInt(N:Vn) else UInt(Vn:N);
7 m = if dp_operation then UInt(M:Vm) else UInt(Vm:M);
```

#### Assembler symbols for all encodings

|                         |                                                                                               |
|-------------------------|-----------------------------------------------------------------------------------------------|
| <code>&lt;q&gt;</code>  | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                      |
| <code>&lt;Sd&gt;</code> | Is the 32-bit name of the floating-point destination register, encoded in the "Vd:D" field.   |
| <code>&lt;Sn&gt;</code> | Is the 32-bit name of the first floating-point source register, encoded in the "Vn:N" field.  |
| <code>&lt;Sm&gt;</code> | Is the 32-bit name of the second floating-point source register, encoded in the "Vm:M" field. |
| <code>&lt;Dd&gt;</code> | Is the 64-bit name of the floating-point destination register, encoded in the "D:Vd" field.   |
| <code>&lt;Dn&gt;</code> | Is the 64-bit name of the first floating-point source register, encoded in the "N:Vn" field.  |
| <code>&lt;Dm&gt;</code> | Is the 64-bit name of the second floating-point source register, encoded in the "M:Vm" field. |

#### Operation for all encodings

```
1 EncodingSpecificOperations();
2 ExecuteFPCheck();
3 if dp_operation then
4 if maximum then
5 D[d] = FPMaxNum(D[n], D[m]);
```

```
6 else
7 D[d] = FPMinNum(D[n], D[m]);
8 else
9 if maximum then
10 S[d] = FPMaxNum(S[n], S[m]);
11 else
12 S[d] = FPMinNum(S[n], S[m]);
```



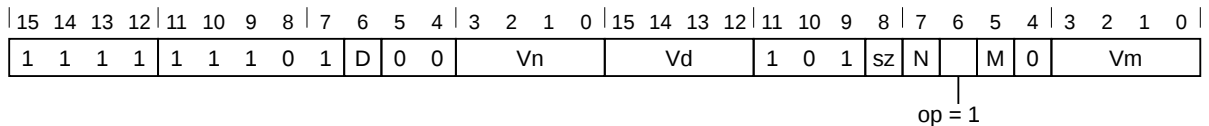
## C2.4.276 VMINNM

Floating-point Minimum Number. Floating-point Minimum Number determines the floating-point minimum number.

NaN handling is specified by IEEE754-2008.

### T2

*Armv8-M Floating-point Extension only*, *sz == 1* UNDEFINED in single-precision only implementations.



#### Single-precision scalar variant

*Armv8-M Floating-point Extension only*.

Applies when **sz == 0**.

```
VMINNM{<q>}.F32 <Sd>, <Sn>, <Sm>
 // Not permitted in IT block
```

#### Double-precision scalar variant

*Armv8-M Floating-point Extension only*.

Applies when **sz == 1**.

```
VMINNM{<q>}.F64 <Dd>, <Dn>, <Dm>
 // Not permitted in IT block
```

#### Decode for this encoding

```
1 dp_operation = (sz == '1');
2 CheckDecodeFaults(dp_operation);
3 if InITBlock() then UNPREDICTABLE;
4 maximum = (op == '0');
5 d = if dp_operation then UInt(D:Vd) else UInt(Vd:D);
6 n = if dp_operation then UInt(N:Vn) else UInt(Vn:N);
7 m = if dp_operation then UInt(M:Vm) else UInt(Vm:M);
```

#### Assembler symbols for all encodings

|                         |                                                                                               |
|-------------------------|-----------------------------------------------------------------------------------------------|
| <code>&lt;q&gt;</code>  | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                      |
| <code>&lt;Sd&gt;</code> | Is the 32-bit name of the floating-point destination register, encoded in the "Vd:D" field.   |
| <code>&lt;Sn&gt;</code> | Is the 32-bit name of the first floating-point source register, encoded in the "Vn:N" field.  |
| <code>&lt;Sm&gt;</code> | Is the 32-bit name of the second floating-point source register, encoded in the "Vm:M" field. |
| <code>&lt;Dd&gt;</code> | Is the 64-bit name of the floating-point destination register, encoded in the "D:Vd" field.   |
| <code>&lt;Dn&gt;</code> | Is the 64-bit name of the first floating-point source register, encoded in the "N:Vn" field.  |
| <code>&lt;Dm&gt;</code> | Is the 64-bit name of the second floating-point source register, encoded in the "M:Vm" field. |

#### Operation for all encodings

```
1 EncodingSpecificOperations();
2 ExecuteFPCheck();
3 if dp_operation then
4 if maximum then
5 D[d] = FPMaxNum(D[n], D[m]);
```

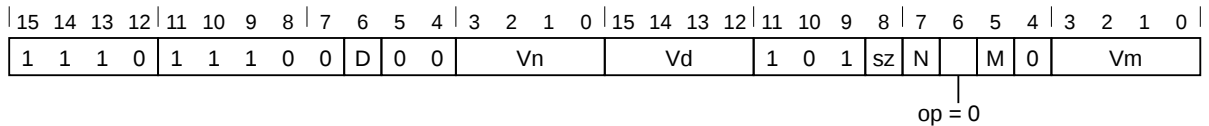
```
6 else
7 D[d] = FPMinNum(D[n], D[m]);
8 else
9 if maximum then
10 S[d] = FPMaxNum(S[n], S[m]);
11 else
12 S[d] = FPMinNum(S[n], S[m]);
```

## C2.4.277 VMLA

Floating-point Multiply Accumulate. Floating-point Multiply Accumulate multiplies two floating-point registers, adds the product to the destination register, and places the result in the destination register.

### T2

Armv8-M Floating-point Extension only, `sz == 1` UNDEFINED in single-precision only implementations.



#### Single-precision scalar variant

Armv8-M Floating-point Extension only.

Applies when `sz == 0`.

VMLA{<c>}{<q>}.F32 <Sd>, <Sn>, <Sm>

#### Double-precision scalar variant

Armv8-M Floating-point Extension only.

Applies when `sz == 1`.

VMLA{<c>}{<q>}.F64 <Dd>, <Dn>, <Dm>

#### Decode for this encoding

```

1 dp_operation = (sz == '1');
2 CheckDecodeFaults(dp_operation);
3 add = (op == '0');
4 d = if dp_operation then UInt(D:Vd) else UInt(Vd:D);
5 n = if dp_operation then UInt(N:Vn) else UInt(Vn:N);
6 m = if dp_operation then UInt(M:Vm) else UInt(Vm:M);

```

#### Assembler symbols for all encodings

|                         |                                                                                               |
|-------------------------|-----------------------------------------------------------------------------------------------|
| <code>&lt;c&gt;</code>  | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                      |
| <code>&lt;q&gt;</code>  | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                      |
| <code>&lt;Sd&gt;</code> | Is the 32-bit name of the floating-point destination register, encoded in the "Vd:D" field.   |
| <code>&lt;Sn&gt;</code> | Is the 32-bit name of the first floating-point source register, encoded in the "Vn:N" field.  |
| <code>&lt;Sm&gt;</code> | Is the 32-bit name of the second floating-point source register, encoded in the "Vm:M" field. |
| <code>&lt;Dd&gt;</code> | Is the 64-bit name of the floating-point destination register, encoded in the "D:Vd" field.   |
| <code>&lt;Dn&gt;</code> | Is the 64-bit name of the first floating-point source register, encoded in the "N:Vn" field.  |
| <code>&lt;Dm&gt;</code> | Is the 64-bit name of the second floating-point source register, encoded in the "M:Vm" field. |

#### Operation for all encodings

```

1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 ExecuteFPCheck();
4 if dp_operation then
5 addend64 = if add then FPMul(D[n], D[m], TRUE) else FPNeg(FPMul(D[n], D[m], TRUE));
6 D[d] = FPAdd(D[d], addend64, TRUE);
7 else
8 addend32 = if add then FPMul(S[n], S[m], TRUE) else FPNeg(FPMul(S[n], S[m], TRUE));
9 S[d] = FPAdd(S[d], addend32, TRUE);

```

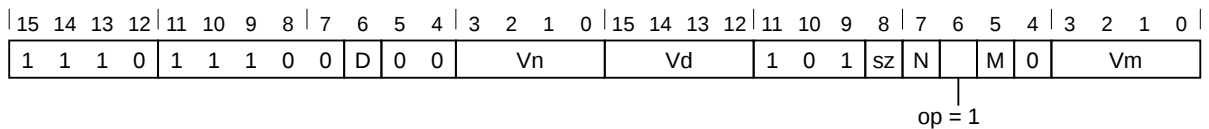
## C2.4.278 VMLS

Floating-point Multiply Subtract. Floating-point Multiply Subtract multiplies two floating-point registers, subtracts the product from the destination floating-point register, and places the result in the destination floating-point register.

Arm recommends that software does not use the **VMLS** instruction in the Round towards +Infinity and Round towards -Infinity rounding modes, because the rounding of the product and of the sum can change the result of the instruction in opposite directions, defeating the purpose of these rounding modes.

### T2

*Armv8-M Floating-point Extension only*, *sz == 1* UNDEFINED in single-precision only implementations.



#### Single-precision scalar variant

*Armv8-M Floating-point Extension only.*

Applies when **sz == 0**.

VMLS {<c>} {<q>} .F32 <Sd>, <Sn>, <Sm>

#### Double-precision scalar variant

*Armv8-M Floating-point Extension only.*

Applies when **sz == 1**.

VMLS {<c>} {<q>} .F64 <Dd>, <Dn>, <Dm>

#### Decode for this encoding

```

1 dp_operation = (sz == '1');
2 CheckDecodeFaults(dp_operation);
3 add = (op == '0');
4 d = if dp_operation then UInt(D:Vd) else UInt(Vd:D);
5 n = if dp_operation then UInt(N:Vn) else UInt(Vn:N);
6 m = if dp_operation then UInt(M:Vm) else UInt(Vm:M);

```

#### Assembler symbols for all encodings

|                   |                                                                                               |
|-------------------|-----------------------------------------------------------------------------------------------|
| <b>&lt;c&gt;</b>  | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                      |
| <b>&lt;q&gt;</b>  | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                      |
| <b>&lt;Sd&gt;</b> | Is the 32-bit name of the floating-point destination register, encoded in the "Vd:D" field.   |
| <b>&lt;Sn&gt;</b> | Is the 32-bit name of the first floating-point source register, encoded in the "Vn:N" field.  |
| <b>&lt;Sm&gt;</b> | Is the 32-bit name of the second floating-point source register, encoded in the "Vm:M" field. |
| <b>&lt;Dd&gt;</b> | Is the 64-bit name of the floating-point destination register, encoded in the "D:Vd" field.   |
| <b>&lt;Dn&gt;</b> | Is the 64-bit name of the first floating-point source register, encoded in the "N:Vn" field.  |
| <b>&lt;Dm&gt;</b> | Is the 64-bit name of the second floating-point source register, encoded in the "M:Vm" field. |

#### Operation for all encodings

```

1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 ExecuteFPCheck();
4 if dp_operation then

```

```
5 addend64 = if add then FPMul (D[n], D[m], TRUE) else FPNeg (FPMul (D[n], D[m], TRUE));
6 D[d] = FPAdd (D[d], addend64, TRUE);
7
8 else
9 addend32 = if add then FPMul (S[n], S[m], TRUE) else FPNeg (FPMul (S[n], S[m], TRUE));
10 S[d] = FPAdd (S[d], addend32, TRUE);
```

### C2.4.279 VMOV (between general-purpose register and single-precision register)

Floating-point Move (between general-purpose register and single-precision register). Floating-point Move (between general-purpose register and single-precision register) transfers the contents of a single-precision register to a general-purpose register, or the contents of a general-purpose register to a single-precision register.

#### T1

Armv8-M Floating-point Extension only

|    |    |    |    |    |    |   |   |   |   |   |    |    |   |   |   |    |    |    |    |    |    |   |   |   |     |     |   |     |     |     |     |     |
|----|----|----|----|----|----|---|---|---|---|---|----|----|---|---|---|----|----|----|----|----|----|---|---|---|-----|-----|---|-----|-----|-----|-----|-----|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4  | 3  | 2 | 1 | 0 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6   | 5   | 4 | 3   | 2   | 1   | 0   |     |
| 1  | 1  | 1  | 0  | 1  | 1  | 1 | 0 | 0 | 0 | 0 | op | Vn |   |   |   | Rt |    |    |    | 1  | 0  | 1 | 0 | N | (0) | (0) | 1 | (0) | (0) | (0) | (0) | (0) |

#### Encoding

Applies when **op == 0**.

VMOV{<c>}{<q>} <Sn>, <Rt>

#### Encoding

Applies when **op == 1**.

VMOV{<c>}{<q>} <Rt>, <Sn>

#### Decode for this encoding

```

1 CheckDecodeFaults();
2 to_arm_register = (op == '1'); t = UInt(Rt); n = UInt(Vn:N);
3 if t == 15 || t == 13 then UNPREDICTABLE;
```

#### Assembler symbols for all encodings

<Rt> Is the general-purpose register that <Sn> will be transferred to or from, encoded in the "Rt" field.  
 <Sn> Is the 32-bit name of the floating-point register to be transferred, encoded in the "Vn:N" field.  
 <c> See [C1.2.5 Standard assembler syntax fields](#) on page 309.  
 <q> See [C1.2.5 Standard assembler syntax fields](#) on page 309.

#### Operation for all encodings

```

1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 ExecuteFPCheck();
4 if to_arm_register then
5 R[t] = S[n];
6 else
7 S[n] = R[t];
```

## C2.4.280 VMOV (between two general-purpose registers and a doubleword register)

Floating-point Move (between two general-purpose registers and a doubleword register). Floating-point Move (between two general-purpose registers and a doubleword register) transfers two words from two general-purpose registers to a doubleword register, or from a doubleword register to two general-purpose registers.

### T1

Armv8-M Floating-point Extension only

|    |    |    |    |    |    |   |   |   |   |   |    |     |   |   |   |    |    |    |    |    |    |   |   |   |   |   |   |    |   |   |   |
|----|----|----|----|----|----|---|---|---|---|---|----|-----|---|---|---|----|----|----|----|----|----|---|---|---|---|---|---|----|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4  | 3   | 2 | 1 | 0 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3  | 2 | 1 | 0 |
| 1  | 1  | 1  | 0  | 1  | 1  | 0 | 0 | 0 | 1 | 0 | op | Rt2 |   |   |   | Rt |    |    |    | 1  | 0  | 1 | 1 | 0 | 0 | M | 1 | Vm |   |   |   |

### Encoding

Applies when `op == 1`.

VMOV{<c>}{<q>} <Rt>, <Rt2>, <Dm>

### Encoding

Applies when `op == 0`.

VMOV{<c>}{<q>} <Dm>, <Rt>, <Rt2>

### Decode for this encoding

```

1 CheckDecodeFaults();
2 to_arm_registers = (op == '1'); t = UInt(Rt); t2 = UInt(Rt2); m = UInt(M:Vm);
3 if t == 15 || t2 == 15 then UNPREDICTABLE;
4 if t == 13 || t2 == 13 then UNPREDICTABLE;
5 if to_arm_registers && t == t2 then UNPREDICTABLE;
```

### CONSTRAINED UNPREDICTABLE behavior

If `to_arm_registers && t == t2`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The value in the destination register is UNKNOWN.

### Assembler symbols for all encodings

|       |                                                                                                                   |
|-------|-------------------------------------------------------------------------------------------------------------------|
| <Dm>  | Is the 64-bit name of the floating-point register to be transferred, encoded in the "M:Vm" field.                 |
| <Rt2> | Is the first general-purpose register that <Dm>[63:32] will be transferred to or from, encoded in the "Rt" field. |
| <Rt>  | Is the first general-purpose register that <Dm>[31:0] will be transferred to or from, encoded in the "Rt" field.  |
| <c>   | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                                          |
| <q>   | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                                          |

### Operation for all encodings

```

1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 ExecuteFPCheck();
4 if to_arm_registers then
5 R[t] = D[m][31:0];
6 R[t2] = D[m][63:32];
```

```
7 else
8 D[m][31:0] = R[t];
9 D[m][63:32] = R[t2];
```



## C2.4.281 VMOV (between two general-purpose registers and two single-precision registers)

Floating-point Move (between two general-purpose registers and two single-precision registers). Floating-point Move (between two general-purpose registers and two single-precision registers) transfers the contents of two consecutively numbered single-precision registers to two general-purpose registers, or the contents of two general-purpose registers to a pair of single-precision registers. The general-purpose registers do not have to be contiguous.

### T1

Armv8-M Floating-point Extension only

|    |    |    |    |    |    |   |   |   |   |   |    |     |   |   |   |    |    |    |    |    |    |   |   |   |   |   |   |    |   |   |   |
|----|----|----|----|----|----|---|---|---|---|---|----|-----|---|---|---|----|----|----|----|----|----|---|---|---|---|---|---|----|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4  | 3   | 2 | 1 | 0 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3  | 2 | 1 | 0 |
| 1  | 1  | 1  | 0  | 1  | 1  | 0 | 0 | 0 | 1 | 0 | op | Rt2 |   |   |   | Rt |    |    |    | 1  | 0  | 1 | 0 | 0 | 0 | M | 1 | Vm |   |   |   |

### Encoding

Applies when `op == 1`.

VMOV{<c>}{<q>} <Rt>, <Rt2>, <Sm>, <Sm1>

### Encoding

Applies when `op == 0`.

VMOV{<c>}{<q>} <Sm>, <Sm1>, <Rt>, <Rt2>

### Decode for this encoding

```

1 CheckDecodeFaults();
2 to_arm_registers = (op == '1'); t = UInt(Rt); t2 = UInt(Rt2); m = UInt(Vm:M);
3 if t == 15 || t2 == 15 || m == 31 then UNPREDICTABLE;
4 if t == 13 || t2 == 13 then UNPREDICTABLE;
5 if to_arm_registers && t == t2 then UNPREDICTABLE;
```

### CONSTRAINED UNPREDICTABLE behavior

If `to_arm_registers && t == t2`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The value in the destination register is UNKNOWN.

### CONSTRAINED UNPREDICTABLE behavior

If `m == 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- One or more of the single-precision registers become UNKNOWN for a move to the single-precision register. The general-purpose registers listed in the instruction become UNKNOWN for a move from the single-precision registers. This behavior does not affect any other general-purpose registers.

### Assembler symbols for all encodings

|       |                                                                                                              |
|-------|--------------------------------------------------------------------------------------------------------------|
| <Rt2> | Is the second general-purpose register that <Sm1> will be transferred to or from, encoded in the "Rt" field. |
| <Rt>  | Is the first general-purpose register that <Sm> will be transferred to or from, encoded in the "Rt" field.   |

|       |                                                                                                                                  |
|-------|----------------------------------------------------------------------------------------------------------------------------------|
| <Sm1> | Is the 32-bit name of the second floating-point register to be transferred. This is the next floating-point register after <Sm>. |
| <Sm>  | Is the 32-bit name of the first floating-point register to be transferred, encoded in the "Vm:M" field.                          |
| <c>   | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                                                         |
| <q>   | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                                                         |

### Operation for all encodings

```

1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 ExecuteFPCheck();
4 if to_arm_registers then
5 R[t] = S[m];
6 R[t2] = S[m+1];
7 else
8 S[m] = R[t];
9 S[m+1] = R[t2];

```

### C2.4.282 VMOV (half of doubleword register to single general-purpose register)

Floating-point Move (half of doubleword register to single general-purpose register). Floating-point Move (half of doubleword register to single general-purpose register) transfers one word from the upper or lower half of a doubleword register to a general-purpose register.

The pseudocode descriptions of the instruction operation convert the doubleword register description into the corresponding single-precision register. For example, D3[1], indicating the upper word of D3, becomes S7.

#### T1

Armv8-M Floating-point Extension only

|    |    |    |    |    |    |   |   |   |   |   |   |    |   |   |   |    |    |    |    |    |    |   |   |   |   |   |   |     |     |     |     |
|----|----|----|----|----|----|---|---|---|---|---|---|----|---|---|---|----|----|----|----|----|----|---|---|---|---|---|---|-----|-----|-----|-----|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3  | 2 | 1 | 0 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3   | 2   | 1   | 0   |
| 1  | 1  | 1  | 0  | 1  | 1  | 1 | 0 | 0 | 0 | H | 1 | Vn |   |   |   | Rt |    |    |    | 1  | 0  | 1 | 1 | N | 0 | 0 | 1 | (0) | (0) | (0) | (0) |

#### T1 variant

VMOV{<c>}{<q>}{.<dt>} <Rt>, <Dn[x]>

#### Decode for this encoding

```

1 CheckDecodeFaults();
2 t = UInt(Rt); n = UInt(N:Vn);
3 upper = (H == '1');
4 if t == 15 || t == 13 then UNPREDICTABLE;
```

#### Assembler symbols for all encodings

<c> See [C1.2.5 Standard assembler syntax fields](#) on page 309.  
 <q> See [C1.2.5 Standard assembler syntax fields](#) on page 309.  
 <dt> The data size. It must be either 32 or omitted.  
 <Dn[x]> The source doubleword register and required word. The register <Dd> is encoded in N:Vn. x is 1 for the top half of the register, or 0 for the bottom half of the register, and is encoded in H.  
 <Rt> The destination general-purpose register, encoded in the "Rt" field.

#### Operation for all encodings

```

1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 ExecuteFPCheck();
4 if upper then
5 R[t] = D[n][63:32];
6 else
7 R[t] = D[n][31:0];
```

### C2.4.283 VMOV (immediate)

Floating-point Move (immediate). Floating-point Move (immediate) places an immediate constant into the destination floating-point register.

#### T2

Armv8-M Floating-point Extension only,  $sz == 1$  UNDEFINED in single-precision only implementations.

|    |    |    |    |    |    |   |   |   |   |   |   |       |   |   |   |    |    |    |    |    |    |   |    |     |   |     |   |       |   |   |   |
|----|----|----|----|----|----|---|---|---|---|---|---|-------|---|---|---|----|----|----|----|----|----|---|----|-----|---|-----|---|-------|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3     | 2 | 1 | 0 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8  | 7   | 6 | 5   | 4 | 3     | 2 | 1 | 0 |
| 1  | 1  | 1  | 0  | 1  | 1  | 1 | 0 | 1 | D | 1 | 1 | imm4H |   |   |   | Vd |    |    |    | 1  | 0  | 1 | sz | (0) | 0 | (0) | 0 | imm4L |   |   |   |

#### Single-precision scalar variant

Armv8-M Floating-point Extension only.

Applies when  $sz == 0$ .

VMOV{<c>}{<q>}.F32 <Sd>, #<imm>

#### Double-precision scalar variant

Armv8-M Floating-point Extension only.

Applies when  $sz == 1$ .

VMOV{<c>}{<q>}.F64 <Dd>, #<imm>

#### Decode for this encoding

```

1 dp_operation = (sz == '1');
2 CheckDecodeFaults(dp_operation);
3 if dp_operation then
4 d = UInt(D:Vd); imm64 = VFPEExpandImm(imm4H:imm4L, 64);
5 else
6 d = UInt(Vd:D); imm32 = VFPEExpandImm(imm4H:imm4L, 32);

```

#### Assembler symbols for all encodings

<c> See [C1.2.5 Standard assembler syntax fields](#) on page 309.  
 <q> See [C1.2.5 Standard assembler syntax fields](#) on page 309.  
 <Sd> Is the 32-bit name of the floating-point destination register, encoded in the "Vd:D" field.  
 <Dd> Is the 64-bit name of the floating-point destination register, encoded in the "D:Vd" field.  
 <imm> Is a floating-point constant. For details of the range of constants available and the encoding of <imm>, see the definition of [VFPEExpandImm\(\)](#).

#### Operation for all encodings

```

1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 ExecuteFPCheck();
4 if dp_operation then
5 D[d] = imm64;
6 else
7 S[d] = imm32;

```

## C2.4.284 VMOV (register)

Floating-point Move (register). Floating-point Move (register) copies the contents of one register to another.

### T2

*Armv8-M Floating-point Extension only*, == 1 UNDEFINED in single-precision only implementations.

|    |    |    |    |    |    |   |   |   |   |   |   |   |   |   |   |    |    |    |    |    |    |   |   |   |    |   |   |   |   |   |   |
|----|----|----|----|----|----|---|---|---|---|---|---|---|---|---|---|----|----|----|----|----|----|---|---|---|----|---|---|---|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6  | 5 | 4 | 3 | 2 | 1 | 0 |
| 1  | 1  | 1  | 0  | 1  | 1  | 1 | 0 | 1 | D | 1 | 1 | 0 | 0 | 0 | 0 | Vd | 1  | 0  | 1  | sz | 0  | 1 | M | 0 | Vm |   |   |   |   |   |   |

#### Single-precision scalar variant

*Armv8-M Floating-point Extension only*.

Applies when **sz == 0**.

VMOV{<c>}{<q>}.F32 <Sd>, <Sm>

#### Double-precision scalar variant

*Armv8-M Floating-point Extension only*.

Applies when **sz == 1**.

VMOV{<c>}{<q>}.F64 <Dd>, <Dm>

#### Decode for this encoding

```

1 dp_operation = (sz == '1');
2 CheckDecodeFaults(dp_operation);
3 d = if dp_operation then UInt(D:Vd) else UInt(Vd:D);
4 m = if dp_operation then UInt(M:Vm) else UInt(Vm:M);

```

#### Assembler symbols for all encodings

|      |                                                                                             |
|------|---------------------------------------------------------------------------------------------|
| <c>  | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                    |
| <q>  | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                    |
| <Sd> | Is the 32-bit name of the floating-point destination register, encoded in the "Vd:D" field. |
| <Sm> | Is the 32-bit name of the floating-point source register, encoded in the "Vm:M" field.      |
| <Dd> | Is the 64-bit name of the floating-point destination register, encoded in the "D:Vd" field. |
| <Dm> | Is the 64-bit name of the floating-point source register, encoded in the "M:Vm" field.      |

#### Operation for all encodings

```

1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 ExecuteFPCheck();
4 if dp_operation then
5 D[d] = D[m];
6 else
7 S[d] = S[m];

```

### C2.4.285 VMOV (single general-purpose register to half of doubleword register)

Floating-point Move (single general-purpose register to half of doubleword register). Floating-point Move (single general-purpose register to half of doubleword register) transfers one word from a general-purpose register to the upper or lower half of a doubleword register.

The pseudocode descriptions of the instruction operation convert the doubleword register description into the corresponding single-precision register. For example, D3[1], indicating the upper word of D3, becomes S7.

#### T1

Armv8-M Floating-point Extension only

|    |    |    |    |    |    |   |   |   |   |   |   |    |   |   |   |    |    |    |    |    |    |   |   |   |   |   |   |     |     |     |     |
|----|----|----|----|----|----|---|---|---|---|---|---|----|---|---|---|----|----|----|----|----|----|---|---|---|---|---|---|-----|-----|-----|-----|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3  | 2 | 1 | 0 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3   | 2   | 1   | 0   |
| 1  | 1  | 1  | 0  | 1  | 1  | 1 | 0 | 0 | 0 | H | 0 | Vd |   |   |   | Rt |    |    |    | 1  | 0  | 1 | 1 | D | 0 | 0 | 1 | (0) | (0) | (0) | (0) |

#### T1 variant

VMOV{<c>}{<q>}{.<size>} <Dd[x]>, <Rt>

#### Decode for this encoding

```

1 CheckDecodeFaults();
2 d = UInt(D:Vd); t = UInt(Rt);
3 upper = (H == '1');
4 if t == 15 || t == 13 then UNPREDICTABLE;
```

#### Assembler symbols for all encodings

<c> See [C1.2.5 Standard assembler syntax fields](#) on page 309.  
 <q> See [C1.2.5 Standard assembler syntax fields](#) on page 309.  
 <size> The data size. It must be either 32 or omitted.  
 <Dd[x]> The destination doubleword register and required word. The register <Dd> is encoded in D:Vd. x is 1 for the top half of the register, or 0 for the bottom half of the register, and is encoded in H.  
 <Rt> The source general-purpose register, encoded in the "Rt" field.

#### Operation for all encodings

```

1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 ExecuteFPCheck();
4 if upper then
5 D[d][63:32] = R[t];
6 else
7 D[d][31:0] = R[t];
```

## C2.4.286 VMRS

Move to general-purpose Register from Floating-point Special register. Move to general-purpose Register from Floating-point Special register moves the value of FPSCR to a general-purpose register, or the values of FPSCR condition flags to the APSR condition flags.

### T1

Armv8-M Floating-point Extension only

|    |    |    |    |    |    |   |   |   |   |   |   |     |     |     |     |    |    |    |    |    |     |     |     |   |     |     |     |     |     |     |     |
|----|----|----|----|----|----|---|---|---|---|---|---|-----|-----|-----|-----|----|----|----|----|----|-----|-----|-----|---|-----|-----|-----|-----|-----|-----|-----|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3   | 2   | 1   | 0   | 15 | 14 | 13 | 12 | 11 | 10  | 9   | 8   | 7 | 6   | 5   | 4   | 3   | 2   | 1   | 0   |
| 1  | 1  | 1  | 0  | 1  | 1  | 1 | 0 | 1 | 1 | 1 | 1 | (0) | (0) | (0) | (1) | Rt | 1  | 0  | 1  | 0  | (0) | (0) | (0) | 1 | (0) | (0) | (0) | (0) | (0) | (0) | (0) |

### T1 variant

VMRS {<c>} {<q>} <Rt>, FPSCR

### Decode for this encoding

```
1 CheckDecodeFaults();
2 t = UInt(Rt);
3 if t == 13 then UNPREDICTABLE;
```

### Assembler symbols for all encodings

<c> See [C1.2.5 Standard assembler syntax fields](#) on page 309.  
 <q> See [C1.2.5 Standard assembler syntax fields](#) on page 309.  
 <Rt> Is the general-purpose destination register, encoded in the "Rt" field. Is one of:  
     APSR\_nzcv Encoded as 0b1111. This instruction transfers the FPSCR.N, Z, C, V condition flags to the APSR.N, Z, C, V condition flags.  
     R0-R14 General-purpose register.

### Operation for all encodings

```
1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 ExecuteFPCheck();
4 SerializeVFP();
5 VFPExcBarrier();
6 if t == 15 then
7 APSR.N = FPSCR.N;
8 APSR.Z = FPSCR.Z;
9 APSR.C = FPSCR.C;
10 APSR.V = FPSCR.V;
11 else
12 R[t] = FPSCR;
```

## C2.4.287 VMSR

Move to Floating-point Special register from general-purpose Register. Move to Floating-point Special register from general-purpose Register moves the value of a general-purpose register to FPSCR.

### T1

Armv8-M Floating-point Extension only

|    |    |    |    |    |    |   |   |   |   |   |   |     |     |     |     |    |    |    |    |    |     |     |     |   |     |     |     |     |   |   |   |
|----|----|----|----|----|----|---|---|---|---|---|---|-----|-----|-----|-----|----|----|----|----|----|-----|-----|-----|---|-----|-----|-----|-----|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3   | 2   | 1   | 0   | 15 | 14 | 13 | 12 | 11 | 10  | 9   | 8   | 7 | 6   | 5   | 4   | 3   | 2 | 1 | 0 |
| 1  | 1  | 1  | 0  | 1  | 1  | 1 | 0 | 1 | 1 | 1 | 0 | (0) | (0) | (0) | (1) | Rt | 1  | 0  | 1  | 0  | (0) | (0) | (0) | 1 | (0) | (0) | (0) | (0) |   |   |   |

### T1 variant

VMSR{<c>}{<q>} FPSCR, <Rt>

### Decode for this encoding

```

1 CheckDecodeFaults();
2 t = UInt(Rt);
3 if t == 15 || t == 13 then UNPREDICTABLE;
```

### Assembler symbols for all encodings

<c> See [C1.2.5 Standard assembler syntax fields](#) on page 309.  
 <q> See [C1.2.5 Standard assembler syntax fields](#) on page 309.  
 <Rt> Is the general-purpose source register to be transferred to FPSCR, encoded in the "Rt" field.

### Operation for all encodings

```

1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 ExecuteFPCheck();
4 SerializeVFP();
5 VFPExcBarrier();
6 FPSCR = R[t];
```



## C2.4.288 VMUL

Floating-point Multiply. Floating-point Multiply multiplies two floating-point register values, and places the result in the destination floating-point register.

### T2

Armv8-M Floating-point Extension only, *sz* == 1 UNDEFINED in single-precision only implementations.

|    |    |    |    |    |    |   |   |   |   |   |   |    |   |   |   |    |    |    |    |    |    |   |    |   |   |   |   |    |   |   |   |
|----|----|----|----|----|----|---|---|---|---|---|---|----|---|---|---|----|----|----|----|----|----|---|----|---|---|---|---|----|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3  | 2 | 1 | 0 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8  | 7 | 6 | 5 | 4 | 3  | 2 | 1 | 0 |
| 1  | 1  | 1  | 0  | 1  | 1  | 1 | 0 | 0 | D | 1 | 0 | Vn |   |   |   | Vd |    |    |    | 1  | 0  | 1 | sz | N | 0 | M | 0 | Vm |   |   |   |

#### Single-precision scalar variant

Armv8-M Floating-point Extension only.

Applies when *sz* == 0.

VMUL{<c>}{<q>}.F32 {<Sd>}, {<Sn>}, <Sm>

#### Double-precision scalar variant

Armv8-M Floating-point Extension only.

Applies when *sz* == 1.

VMUL{<c>}{<q>}.F64 {<Dd>}, {<Dn>}, <Dm>

#### Decode for this encoding

```

1 dp_operation = (sz == '1');
2 CheckDecodeFaults(dp_operation);
3 d = if dp_operation then UInt(D:Vd) else UInt(Vd:D);
4 n = if dp_operation then UInt(N:Vn) else UInt(Vn:N);
5 m = if dp_operation then UInt(M:Vm) else UInt(Vm:M);

```

#### Assembler symbols for all encodings

|      |                                                                                               |
|------|-----------------------------------------------------------------------------------------------|
| <c>  | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                      |
| <q>  | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                      |
| <Sd> | Is the 32-bit name of the floating-point destination register, encoded in the "Vd:D" field.   |
| <Sn> | Is the 32-bit name of the first floating-point source register, encoded in the "Vn:N" field.  |
| <Sm> | Is the 32-bit name of the second floating-point source register, encoded in the "Vm:M" field. |
| <Dd> | Is the 64-bit name of the floating-point destination register, encoded in the "D:Vd" field.   |
| <Dn> | Is the 64-bit name of the first floating-point source register, encoded in the "N:Vn" field.  |
| <Dm> | Is the 64-bit name of the second floating-point source register, encoded in the "M:Vm" field. |

#### Operation for all encodings

```

1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 ExecuteFPCheck();
4 if dp_operation then
5 D[d] = FPMul(D[n], D[m], TRUE);
6 else
7 S[d] = FPMul(S[n], S[m], TRUE);

```

## C2.4.289 VNEG

Floating-point Negate. Floating-point Negate inverts the sign bit of a single-precision or double-precision register, and places the result in the destination register.

### T2

*Armv8-M Floating-point Extension only*, *sz == 1 UNDEFINED in single-precision only implementations.*

|    |    |    |    |    |    |   |   |   |   |   |   |   |   |   |   |    |    |    |    |    |    |   |   |   |   |   |   |   |   |   |    |
|----|----|----|----|----|----|---|---|---|---|---|---|---|---|---|---|----|----|----|----|----|----|---|---|---|---|---|---|---|---|---|----|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0  |
| 1  | 1  | 1  | 0  | 1  | 1  | 1 | 0 | 1 | D | 1 | 1 | 0 | 0 | 0 | 1 | Vd | 1  | 0  | 1  | sz | 0  | 1 | M | 0 |   |   |   |   |   |   | Vm |

#### Single-precision scalar variant

*Armv8-M Floating-point Extension only.*

Applies when **sz == 0**.

VNEG{<c>}{<q>}.F32 <Sd>, <Sm>

#### Double-precision scalar variant

*Armv8-M Floating-point Extension only.*

Applies when **sz == 1**.

VNEG{<c>}{<q>}.F64 <Dd>, <Dm>

#### Decode for this encoding

```

1 dp_operation = (sz == '1');
2 CheckDecodeFaults(dp_operation);
3 d = if dp_operation then UInt(D:Vd) else UInt(Vd:D);
4 m = if dp_operation then UInt(M:Vm) else UInt(Vm:M);

```

#### Assembler symbols for all encodings

|      |                                                                                             |
|------|---------------------------------------------------------------------------------------------|
| <c>  | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                    |
| <q>  | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                    |
| <Sd> | Is the 32-bit name of the floating-point destination register, encoded in the "Vd:D" field. |
| <Sm> | Is the 32-bit name of the floating-point source register, encoded in the "Vm:M" field.      |
| <Dd> | Is the 64-bit name of the floating-point destination register, encoded in the "D:Vd" field. |
| <Dm> | Is the 64-bit name of the floating-point source register, encoded in the "M:Vm" field.      |

#### Operation for all encodings

```

1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 ExecuteFPCheck();
4 if dp_operation then
5 D[d] = FPNeg(D[m]);
6 else
7 S[d] = FPNeg(S[m]);

```

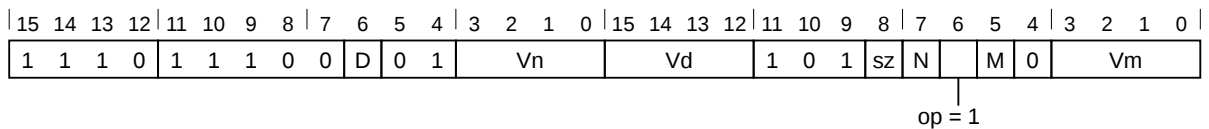
## C2.4.290 VNMLA

Floating-point Multiply Accumulate and Negate. Floating-point Multiply Accumulate and Negate multiplies two floating-point register values, adds the negation of the floating-point value in the destination register to the negation of the product, and writes the result back to the destination register.

Arm recommends that software does not use the [VNMLA](#) instruction in the Round towards +Infinity and Round towards -Infinity rounding modes, because the rounding of the product and of the sum can change the result of the instruction in opposite directions, defeating the purpose of these rounding modes.

### T1

*Armv8-M Floating-point Extension only*, *sz == 1* UNDEFINED in single-precision only implementations.



#### Single-precision scalar variant

*Armv8-M Floating-point Extension only.*

Applies when **sz == 0**.

VNMLA{<c>} {<q>} .F32 <Sd>, <Sn>, <Sm>

#### Double-precision scalar variant

*Armv8-M Floating-point Extension only.*

Applies when **sz == 1**.

VNMLA{<c>} {<q>} .F64 <Dd>, <Dn>, <Dm>

#### Decode for this encoding

```

1 dp_operation = (sz == '1');
2 CheckDecodeFaults(dp_operation);
3 type = if op == '1' then VFPNegMul_VNMLA else VFPNegMul_VNMLS;
4 d = if dp_operation then UInt(D:Vd) else UInt(Vd:D);
5 n = if dp_operation then UInt(N:Vn) else UInt(Vn:N);
6 m = if dp_operation then UInt(M:Vm) else UInt(Vm:M);

```

#### Assembler symbols for all encodings

|                         |                                                                                               |
|-------------------------|-----------------------------------------------------------------------------------------------|
| <code>&lt;c&gt;</code>  | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                      |
| <code>&lt;q&gt;</code>  | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                      |
| <code>&lt;Sd&gt;</code> | Is the 32-bit name of the floating-point destination register, encoded in the "Vd:D" field.   |
| <code>&lt;Sn&gt;</code> | Is the 32-bit name of the first floating-point source register, encoded in the "Vn:N" field.  |
| <code>&lt;Sm&gt;</code> | Is the 32-bit name of the second floating-point source register, encoded in the "Vm:M" field. |
| <code>&lt;Dd&gt;</code> | Is the 64-bit name of the floating-point destination register, encoded in the "D:Vd" field.   |
| <code>&lt;Dn&gt;</code> | Is the 64-bit name of the first floating-point source register, encoded in the "N:Vn" field.  |
| <code>&lt;Dm&gt;</code> | Is the 64-bit name of the second floating-point source register, encoded in the "M:Vm" field. |

#### Operation for all encodings

```

1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 ExecuteFPCheck();
4 if dp_operation then

```

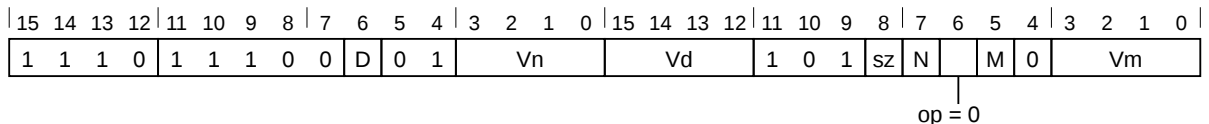
```
5 product64 = FPMul(D[n], D[m], TRUE);
6 case type of
7 when VFPNegMul_VNMLA D[d] = FAdd(FPNeg(D[d]), FPNeg(product64), TRUE);
8 when VFPNegMul_VNMLS D[d] = FAdd(FPNeg(D[d]), product64, TRUE);
9 when VFPNegMul_VNMUL D[d] = FPNeg(product64);
10 else
11 product32 = FPMul(S[n], S[m], TRUE);
12 case type of
13 when VFPNegMul_VNMLA S[d] = FAdd(FPNeg(S[d]), FPNeg(product32), TRUE);
14 when VFPNegMul_VNMLS S[d] = FAdd(FPNeg(S[d]), product32, TRUE);
15 when VFPNegMul_VNMUL S[d] = FPNeg(product32);
```

## C2.4.291 VNMLS

Floating-point Multiply Subtract and Negate. Floating-point Multiply Subtract and Negate multiplies two floating-point register values, adds the negation of the floating-point value in the destination register to the product, and writes the result back to the destination register.

### T1

Armv8-M Floating-point Extension only, *sz* == 1 UNDEFINED in single-precision only implementations.



#### Single-precision scalar variant

Armv8-M Floating-point Extension only.

Applies when *sz* == 0.

VNMLS {<c>} {<q>} .F32 <Sd>, <Sn>, <Sm>

#### Double-precision scalar variant

Armv8-M Floating-point Extension only.

Applies when *sz* == 1.

VNMLS {<c>} {<q>} .F64 <Dd>, <Dn>, <Dm>

#### Decode for this encoding

```

1 dp_operation = (sz == '1');
2 CheckDecodeFaults(dp_operation);
3 type = if op == '1' then VFPNegMul_VNMLA else VFPNegMul_VNMLS;
4 d = if dp_operation then UInt(D:Vd) else UInt(Vd:D);
5 n = if dp_operation then UInt(N:Vn) else UInt(Vn:N);
6 m = if dp_operation then UInt(M:Vm) else UInt(Vm:M);

```

#### Assembler symbols for all encodings

|                         |                                                                                               |
|-------------------------|-----------------------------------------------------------------------------------------------|
| <code>&lt;c&gt;</code>  | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                      |
| <code>&lt;q&gt;</code>  | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                      |
| <code>&lt;Sd&gt;</code> | Is the 32-bit name of the floating-point destination register, encoded in the "Vd:D" field.   |
| <code>&lt;Sn&gt;</code> | Is the 32-bit name of the first floating-point source register, encoded in the "Vn:N" field.  |
| <code>&lt;Sm&gt;</code> | Is the 32-bit name of the second floating-point source register, encoded in the "Vm:M" field. |
| <code>&lt;Dd&gt;</code> | Is the 64-bit name of the floating-point destination register, encoded in the "D:Vd" field.   |
| <code>&lt;Dn&gt;</code> | Is the 64-bit name of the first floating-point source register, encoded in the "N:Vn" field.  |
| <code>&lt;Dm&gt;</code> | Is the 64-bit name of the second floating-point source register, encoded in the "M:Vm" field. |

#### Operation for all encodings

```

1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 ExecuteFPCheck();
4 if dp_operation then
5 product64 = FPMul(D[n], D[m], TRUE);
6 case type of
7 when VFPNegMul_VNMLA D[d] = FAdd(FPNeg(D[d]), FPNeg(product64), TRUE);
8 when VFPNegMul_VNMLS D[d] = FAdd(FPNeg(D[d]), product64, TRUE);

```

```
9 when VFPNegMul_VNMUL D[d] = FPNeg(product64);
10 else
11 product32 = FPMul(S[n], S[m], TRUE);
12 case type of
13 when VFPNegMul_VNMLA S[d] = FAdd(FPNeg(S[d]), FPNeg(product32), TRUE);
14 when VFPNegMul_VNMLS S[d] = FAdd(FPNeg(S[d]), product32, TRUE);
15 when VFPNegMul_VNMUL S[d] = FPNeg(product32);
```

## C2.4.292 VNMUL

Floating-point Multiply and Negate. Floating-point Multiply and Negate multiplies two floating-point register values, and writes the negation of the result to the destination register.

### T2

Armv8-M Floating-point Extension only,  $sz == 1$  UNDEFINED in single-precision only implementations.

|    |    |    |    |    |    |   |   |   |   |   |   |    |   |   |   |    |    |    |    |    |    |   |    |   |   |   |   |    |   |   |   |
|----|----|----|----|----|----|---|---|---|---|---|---|----|---|---|---|----|----|----|----|----|----|---|----|---|---|---|---|----|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3  | 2 | 1 | 0 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8  | 7 | 6 | 5 | 4 | 3  | 2 | 1 | 0 |
| 1  | 1  | 1  | 0  | 1  | 1  | 1 | 0 | 0 | D | 1 | 0 | Vn |   |   |   | Vd |    |    |    | 1  | 0  | 1 | sz | N | 1 | M | 0 | Vm |   |   |   |

#### Single-precision scalar variant

Armv8-M Floating-point Extension only.

Applies when  $sz == 0$ .

VNMUL{<c>}{<q>}.F32 {<Sd>}, {<Sn>}, {<Sm>}

#### Double-precision scalar variant

Armv8-M Floating-point Extension only.

Applies when  $sz == 1$ .

VNMUL{<c>}{<q>}.F64 {<Dd>}, {<Dn>}, {<Dm>}

#### Decode for this encoding

```

1 dp_operation = (sz == '1');
2 CheckDecodeFaults(dp_operation);
3 type = VFPNegMul_VNMUL;
4 d = if dp_operation then UInt(D:Vd) else UInt(Vd:D);
5 n = if dp_operation then UInt(N:Vn) else UInt(Vn:N);
6 m = if dp_operation then UInt(M:Vm) else UInt(Vm:M);

```

#### Assembler symbols for all encodings

|      |                                                                                               |
|------|-----------------------------------------------------------------------------------------------|
| <c>  | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                      |
| <q>  | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                      |
| <Sd> | Is the 32-bit name of the floating-point destination register, encoded in the "Vd:D" field.   |
| <Sn> | Is the 32-bit name of the first floating-point source register, encoded in the "Vn:N" field.  |
| <Sm> | Is the 32-bit name of the second floating-point source register, encoded in the "Vm:M" field. |
| <Dd> | Is the 64-bit name of the floating-point destination register, encoded in the "D:Vd" field.   |
| <Dn> | Is the 64-bit name of the first floating-point source register, encoded in the "N:Vn" field.  |
| <Dm> | Is the 64-bit name of the second floating-point source register, encoded in the "M:Vm" field. |

#### Operation for all encodings

```

1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 ExecuteFPCheck();
4 if dp_operation then
5 product64 = FPMul(D[n], D[m], TRUE);
6 case type of
7 when VFPNegMul_VNMLA D[d] = FPAAdd(FPNeg(D[d]), FPNeg(product64), TRUE);
8 when VFPNegMul_VNMLS D[d] = FPAAdd(FPNeg(D[d]), product64, TRUE);
9 when VFPNegMul_VNMUL D[d] = FPNeg(product64);
10 else
11 product32 = FPMul(S[n], S[m], TRUE);
12 case type of

```

Chapter C2. Instruction Specification

C2.4. Alphabetical list of instructions

```
13 when VFPNegMul_VNMLA S[d] = FPAdd(FPNeg(S[d]), FPNeg(product32), TRUE);
14 when VFPNegMul_VNMLS S[d] = FPAdd(FPNeg(S[d]), product32, TRUE);
15 when VFPNegMul_VNMUL S[d] = FPNeg(product32);
```



### C2.4.293 VPOP

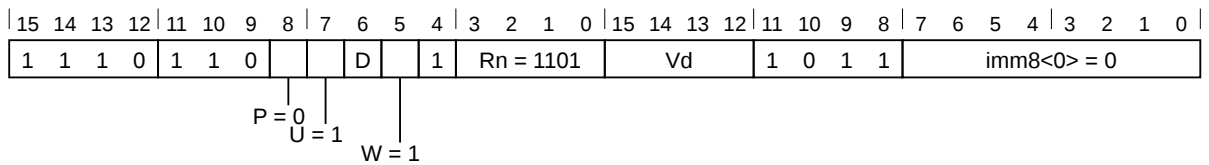
Pop Floating-point registers from stack. Pop Floating-point registers from stack loads multiple consecutive Floating-point registers from the stack.

This instruction is an alias of the [VLDM](#) instruction. This means that:

- The encodings in this description are named to match the encodings of [VLDM](#).
- The description of [VLDM](#) gives the operational pseudocode for this instruction.

#### T1

*Armv8-M Floating-point Extension only*



#### Increment After variant

VPOP{<c>}{<q>}{.<size>} <dreglist>

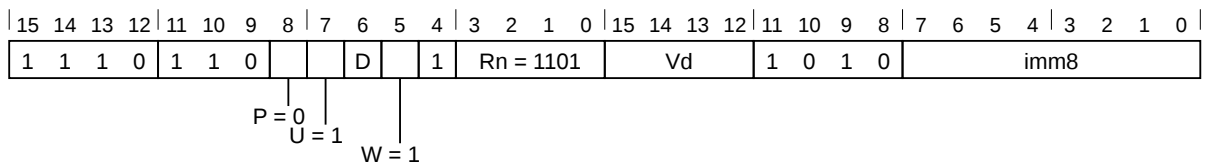
#### is equivalent to

VLDM{<c>}{<q>}{.<size>} SP!, <dreglist>

**and is always the preferred disassembly.**

#### T2

*Armv8-M Floating-point Extension only*



#### Increment After variant

VPOP{<c>}{<q>}{.<size>} <sreglist>

#### is equivalent to

VLDM{<c>}{<q>}{.<size>} SP!, <sreglist>

**and is always the preferred disassembly.**

#### Assembler symbols for all encodings

- <c> See [C1.2.5 Standard assembler syntax fields](#) on page 309.
- <q> See [C1.2.5 Standard assembler syntax fields](#) on page 309.
- <size> An optional data size specifier. If present, it must be equal to the size in bits, 32 or 64, of the registers being transferred.
- <sreglist> Is the list of consecutively numbered 32-bit floating-point registers to be transferred. The first register in the list is encoded in "Vd:D", and "imm8" is set to the number of registers in the list. The list must contain at least one register.
- <dreglist> Is the list of consecutively numbered 64-bit floating-point registers to be transferred. The first register in the list is encoded in "D:Vd", and "imm8" is set to twice the number of registers in the list. The list must contain at least one register, and must not contain more than 16 registers.

### **Operation for all encodings**

The description of [VLDM](#) gives the operational pseudocode for this instruction.

## C2.4.294 V PUSH

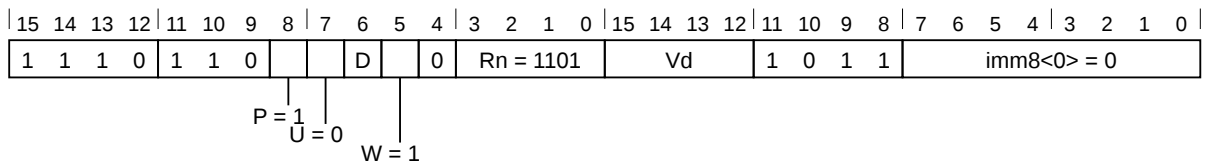
Push Floating-point registers to stack. Push Floating-point registers to stack stores multiple consecutive registers from the Floating-point register file to the stack.

This instruction is an alias of the **VSTM** instruction. This means that:

- The encodings in this description are named to match the encodings of **VSTM**.
- The description of **VSTM** gives the operational pseudocode for this instruction.

### T1

*Armv8-M Floating-point Extension only*



#### Decrement Before variant

V PUSH{<c>}{<q>}{.<size>} <dreglist>

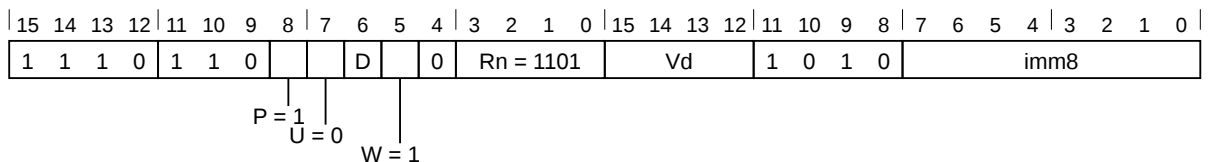
#### is equivalent to

VSTMDB{<c>}{<q>}{.<size>} SP!, <dreglist>

**and is always the preferred disassembly.**

### T2

*Armv8-M Floating-point Extension only*



#### Decrement Before variant

V PUSH{<c>}{<q>}{.<size>} <sreglist>

#### is equivalent to

VSTMDB{<c>}{<q>}{.<size>} SP!, <sreglist>

**and is always the preferred disassembly.**

### Assembler symbols for all encodings

- <c> See [C1.2.5 Standard assembler syntax fields](#) on page 309.
- <q> See [C1.2.5 Standard assembler syntax fields](#) on page 309.
- <size> An optional data size specifier. If present, it must be equal to the size in bits, 32 or 64, of the registers being transferred.
- <sreglist> Is the list of consecutively numbered 32-bit floating-point registers to be transferred. The first register in the list is encoded in "Vd:D", and "imm8" is set to the number of registers in the list. The list must contain at least one register.
- <dreglist> Is the list of consecutively numbered 64-bit floating-point registers to be transferred. The first register in the list is encoded in "D:Vd", and "imm8" is set to twice the number of registers in the list. The list must contain at least one register, and must not contain more than 16 registers.

### **Operation for all encodings**

The description of [VSTM](#) gives the operational pseudocode for this instruction.

## C2.4.295 VRINTA

Floating-point Round to Nearest Integer with Ties to Away. Floating-point Round to Nearest Integer with Ties to Away rounds a floating-point value to an integral floating-point value of the same size using the Round to Nearest with Ties to Away rounding mode. A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

### T1

*Armv8-M Floating-point Extension only*,  $sz == 1$  UNDEFINED in single-precision only implementations.

|    |    |    |    |    |    |   |   |   |   |   |   |   |   |         |   |    |    |    |    |    |    |   |    |   |   |   |   |   |   |    |   |
|----|----|----|----|----|----|---|---|---|---|---|---|---|---|---------|---|----|----|----|----|----|----|---|----|---|---|---|---|---|---|----|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1       | 0 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8  | 7 | 6 | 5 | 4 | 3 | 2 | 1  | 0 |
| 1  | 1  | 1  | 1  | 1  | 1  | 1 | 0 | 1 | D | 1 | 1 | 1 | 0 | RM = 00 |   | Vd |    |    |    | 1  | 0  | 1 | sz | 0 | 1 | M | 0 |   |   | Vm |   |

#### Single-precision scalar variant

*Armv8-M Floating-point Extension only.*

Applies when  $sz == 0$ .

VRINTA{<q>}.F32.F32 <Sd>, <Sm>

#### Double-precision scalar variant

*Armv8-M Floating-point Extension only.*

Applies when  $sz == 1$ .

VRINTA{<q>}.F64.F64 <Dd>, <Dm>

#### Decode for this encoding

```

1 dp_operation = (sz == '1');
2 CheckDecodeFaults(dp_operation);
3 if InitBlock() then UNPREDICTABLE;
4 case RM of
5 when '00' // Round to nearest, with ties away
6 rmode = '01'; away = TRUE;
7 when '01' // Round to nearest, with ties to even
8 rmode = '00'; away = FALSE;
9 when '10' // Round towards Plus Infinity
10 rmode = '01'; away = FALSE;
11 when '11' // Round towards Minus Infinity
12 rmode = '10'; away = FALSE;
13 d = if dp_operation then UInt(D:Vd) else UInt(Vd:D);
14 m = if dp_operation then UInt(M:Vm) else UInt(Vm:M);

```

#### Assembler symbols for all encodings

|      |                                                                                             |
|------|---------------------------------------------------------------------------------------------|
| <q>  | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                    |
| <Sd> | Is the 32-bit name of the floating-point destination register, encoded in the "Vd:D" field. |
| <Sm> | Is the 32-bit name of the floating-point source register, encoded in the "Vm:M" field.      |
| <Dd> | Is the 64-bit name of the floating-point destination register, encoded in the "D:Vd" field. |
| <Dm> | Is the 64-bit name of the floating-point source register, encoded in the "M:Vm" field.      |

#### Operation for all encodings

```

1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 ExecuteFPCheck();
4
5 exact = FALSE;

```

```
6
7 if dp_operation then
8 D[d] = FPRoundInt (D[m], rmode, away, exact);
9 else
10 S[d] = FPRoundInt (S[m], rmode, away, exact);
```

## C2.4.296 VRINTM

Floating-point Round to Integer towards -Infinity. Floating-point Round to Integer towards -Infinity rounds a floating-point value to an integral floating-point value of the same size using the Round towards -Infinity rounding mode. A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

### T1

*Armv8-M Floating-point Extension only*, `sz == 1` UNDEFINED in single-precision only implementations.

|    |    |    |    |    |    |   |   |   |   |   |   |   |   |         |   |    |    |    |    |    |    |   |    |   |   |   |   |   |   |    |   |
|----|----|----|----|----|----|---|---|---|---|---|---|---|---|---------|---|----|----|----|----|----|----|---|----|---|---|---|---|---|---|----|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1       | 0 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8  | 7 | 6 | 5 | 4 | 3 | 2 | 1  | 0 |
| 1  | 1  | 1  | 1  | 1  | 1  | 1 | 0 | 1 | D | 1 | 1 | 1 | 0 | RM = 11 |   | Vd |    |    |    | 1  | 0  | 1 | sz | 0 | 1 | M | 0 |   |   | Vm |   |

#### Single-precision scalar variant

*Armv8-M Floating-point Extension only*.

Applies when `sz == 0`.

`VRINTM{<q>}.F32.F32 <Sd>, <Sm>`

#### Double-precision scalar variant

*Armv8-M Floating-point Extension only*.

Applies when `sz == 1`.

`VRINTM{<q>}.F64.F64 <Dd>, <Dm>`

#### Decode for this encoding

```

1 dp_operation = (sz == '1');
2 CheckDecodeFaults(dp_operation);
3 if InitBlock() then UNPREDICTABLE;
4 case RM of
5 when '00' // Round to nearest, with ties away
6 rmode = '01'; away = TRUE;
7 when '01' // Round to nearest, with ties to even
8 rmode = '00'; away = FALSE;
9 when '10' // Round towards Plus Infinity
10 rmode = '01'; away = FALSE;
11 when '11' // Round towards Minus Infinity
12 rmode = '10'; away = FALSE;
13 d = if dp_operation then UInt(D:Vd) else UInt(Vd:D);
14 m = if dp_operation then UInt(M:Vm) else UInt(Vm:M);

```

#### Assembler symbols for all encodings

|      |                                                                                             |
|------|---------------------------------------------------------------------------------------------|
| <q>  | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                    |
| <Sd> | Is the 32-bit name of the floating-point destination register, encoded in the "Vd:D" field. |
| <Sm> | Is the 32-bit name of the floating-point source register, encoded in the "Vm:M" field.      |
| <Dd> | Is the 64-bit name of the floating-point destination register, encoded in the "D:Vd" field. |
| <Dm> | Is the 64-bit name of the floating-point source register, encoded in the "M:Vm" field.      |

#### Operation for all encodings

```

1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 ExecuteFPCheck();
4
5 exact = FALSE;

```

```
6
7 if dp_operation then
8 D[d] = FPRoundInt (D[m], rmode, away, exact);
9 else
10 S[d] = FPRoundInt (S[m], rmode, away, exact);
```



## C2.4.297 VRINTN

Floating-point Round to Nearest Integer with Ties to Even. Floating-point Round to Nearest Integer with Ties to Even rounds a floating-point value to an integral floating-point value of the same size using the Round to Nearest rounding mode. A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

### T1

*Armv8-M Floating-point Extension only*,  $sz == 1$  UNDEFINED in single-precision only implementations.

|    |    |    |    |    |    |   |   |   |   |   |   |   |   |         |   |    |    |    |    |    |    |   |    |   |   |   |   |   |   |    |   |
|----|----|----|----|----|----|---|---|---|---|---|---|---|---|---------|---|----|----|----|----|----|----|---|----|---|---|---|---|---|---|----|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1       | 0 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8  | 7 | 6 | 5 | 4 | 3 | 2 | 1  | 0 |
| 1  | 1  | 1  | 1  | 1  | 1  | 1 | 0 | 1 | D | 1 | 1 | 1 | 0 | RM = 01 |   | Vd |    |    |    | 1  | 0  | 1 | sz | 0 | 1 | M | 0 |   |   | Vm |   |

#### Single-precision scalar variant

*Armv8-M Floating-point Extension only.*

Applies when  $sz == 0$ .

VRINTN{<q>}.F32.F32 <Sd>, <Sm>

#### Double-precision scalar variant

*Armv8-M Floating-point Extension only.*

Applies when  $sz == 1$ .

VRINTN{<q>}.F64.F64 <Dd>, <Dm>

#### Decode for this encoding

```

1 dp_operation = (sz == '1');
2 CheckDecodeFaults(dp_operation);
3 if InitBlock() then UNPREDICTABLE;
4 case RM of
5 when '00' // Round to nearest, with ties away
6 rmode = '01'; away = TRUE;
7 when '01' // Round to nearest, with ties to even
8 rmode = '00'; away = FALSE;
9 when '10' // Round towards Plus Infinity
10 rmode = '01'; away = FALSE;
11 when '11' // Round towards Minus Infinity
12 rmode = '10'; away = FALSE;
13 d = if dp_operation then UInt(D:Vd) else UInt(Vd:D);
14 m = if dp_operation then UInt(M:Vm) else UInt(Vm:M);

```

#### Assembler symbols for all encodings

|      |                                                                                             |
|------|---------------------------------------------------------------------------------------------|
| <q>  | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                    |
| <Sd> | Is the 32-bit name of the floating-point destination register, encoded in the "Vd:D" field. |
| <Sm> | Is the 32-bit name of the floating-point source register, encoded in the "Vm:M" field.      |
| <Dd> | Is the 64-bit name of the floating-point destination register, encoded in the "D:Vd" field. |
| <Dm> | Is the 64-bit name of the floating-point source register, encoded in the "M:Vm" field.      |

#### Operation for all encodings

```

1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 ExecuteFPCheck();
4
5 exact = FALSE;

```

```
6
7 if dp_operation then
8 D[d] = FPRoundInt (D[m], rmode, away, exact);
9 else
10 S[d] = FPRoundInt (S[m], rmode, away, exact);
```

## C2.4.298 VRINTP

Floating-point Round to Integer towards +Infinity. Floating-point Round to Integer towards +Infinity rounds a floating-point value to an integral floating-point value of the same size using the Round towards +Infinity rounding mode. A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

### T1

*Armv8-M Floating-point Extension only*, `sz == 1` UNDEFINED in single-precision only implementations.

|    |    |    |    |    |    |   |   |   |   |   |   |   |   |         |   |    |    |    |    |    |    |   |    |   |   |   |   |   |   |    |   |
|----|----|----|----|----|----|---|---|---|---|---|---|---|---|---------|---|----|----|----|----|----|----|---|----|---|---|---|---|---|---|----|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1       | 0 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8  | 7 | 6 | 5 | 4 | 3 | 2 | 1  | 0 |
| 1  | 1  | 1  | 1  | 1  | 1  | 1 | 0 | 1 | D | 1 | 1 | 1 | 0 | RM = 10 |   | Vd |    |    |    | 1  | 0  | 1 | sz | 0 | 1 | M | 0 |   |   | Vm |   |

#### Single-precision scalar variant

*Armv8-M Floating-point Extension only*.

Applies when `sz == 0`.

`VRINTP{<q>}.F32.F32 <Sd>, <Sm>`

#### Double-precision scalar variant

*Armv8-M Floating-point Extension only*.

Applies when `sz == 1`.

`VRINTP{<q>}.F64.F64 <Dd>, <Dm>`

#### Decode for this encoding

```

1 dp_operation = (sz == '1');
2 CheckDecodeFaults(dp_operation);
3 if InitBlock() then UNPREDICTABLE;
4 case RM of
5 when '00' // Round to nearest, with ties away
6 rmode = '01'; away = TRUE;
7 when '01' // Round to nearest, with ties to even
8 rmode = '00'; away = FALSE;
9 when '10' // Round towards Plus Infinity
10 rmode = '01'; away = FALSE;
11 when '11' // Round towards Minus Infinity
12 rmode = '10'; away = FALSE;
13 d = if dp_operation then UInt(D:Vd) else UInt(Vd:D);
14 m = if dp_operation then UInt(M:Vm) else UInt(Vm:M);

```

#### Assembler symbols for all encodings

|      |                                                                                             |
|------|---------------------------------------------------------------------------------------------|
| <q>  | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                    |
| <Sd> | Is the 32-bit name of the floating-point destination register, encoded in the "Vd:D" field. |
| <Sm> | Is the 32-bit name of the floating-point source register, encoded in the "Vm:M" field.      |
| <Dd> | Is the 64-bit name of the floating-point destination register, encoded in the "D:Vd" field. |
| <Dm> | Is the 64-bit name of the floating-point source register, encoded in the "M:Vm" field.      |

#### Operation for all encodings

```

1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 ExecuteFPCheck();
4
5 exact = FALSE;

```

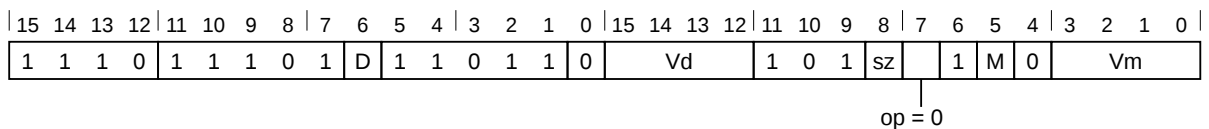
```
6
7 if dp_operation then
8 D[d] = FPRoundInt (D[m], rmode, away, exact);
9 else
10 S[d] = FPRoundInt (S[m], rmode, away, exact);
```

## C2.4.299 VRINTR

Floating-point Round to Integer. Floating-point Round to Integer rounds a floating-point value to an integral floating-point value of the same size using the rounding mode specified in FPSCR. A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

### T1

*Armv8-M Floating-point Extension only*,  $sz == 1$  UNDEFINED in single-precision only implementations.



#### Single-precision scalar variant

*Armv8-M Floating-point Extension only.*

Applies when  $sz == 0$ .

VRINTR{<c>} {<q>} .F32.F32 <Sd>, <Sm>

#### Double-precision scalar variant

*Armv8-M Floating-point Extension only.*

Applies when  $sz == 1$ .

VRINTR{<c>} {<q>} .F64.F64 <Dd>, <Dm>

#### Decode for this encoding

```

1 dp_operation = (sz == '1');
2 CheckDecodeFaults(dp_operation);
3 d = if dp_operation then UInt(D:Vd) else UInt(Vd:D);
4 m = if dp_operation then UInt(M:Vm) else UInt(Vm:M);

```

#### Assembler symbols for all encodings

|                         |                                                                                             |
|-------------------------|---------------------------------------------------------------------------------------------|
| <code>&lt;c&gt;</code>  | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                    |
| <code>&lt;q&gt;</code>  | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                    |
| <code>&lt;Sd&gt;</code> | Is the 32-bit name of the floating-point destination register, encoded in the "Vd:D" field. |
| <code>&lt;Sm&gt;</code> | Is the 32-bit name of the floating-point source register, encoded in the "Vm:M" field.      |
| <code>&lt;Dd&gt;</code> | Is the 64-bit name of the floating-point destination register, encoded in the "D:Vd" field. |
| <code>&lt;Dm&gt;</code> | Is the 64-bit name of the floating-point source register, encoded in the "M:Vm" field.      |

#### Operation for all encodings

```

1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 ExecuteFPCheck();
4
5 rmode = if op == '1' then '11' else FPSCR[23:22];
6 exact = FALSE;
7 away = FALSE;
8
9 if dp_operation then
10 D[d] = FPRoundInt(D[m], rmode, away, exact);
11 else
12 S[d] = FPRoundInt(S[m], rmode, away, exact);

```

### C2.4.300 VRINTX

Floating-point Round to Integer, raising Inexact exception. This instruction rounds a floating-point value to an integral floating-point value of the same size. A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

**VRINTX** uses the rounding mode specified in FPSCR, and raises an Inexact exception when the result value is not numerically equal to the input value.

#### T1

*Armv8-M Floating-point Extension only, sz == 1 UNDEFINED in single-precision only implementations.*

|    |    |    |    |    |    |   |   |   |   |   |   |   |   |   |   |    |    |    |    |    |    |   |   |   |    |   |   |   |   |   |   |
|----|----|----|----|----|----|---|---|---|---|---|---|---|---|---|---|----|----|----|----|----|----|---|---|---|----|---|---|---|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6  | 5 | 4 | 3 | 2 | 1 | 0 |
| 1  | 1  | 1  | 0  | 1  | 1  | 1 | 0 | 1 | D | 1 | 1 | 0 | 1 | 1 | 1 | Vd | 1  | 0  | 1  | sz | 0  | 1 | M | 0 | Vm |   |   |   |   |   |   |

#### Single-precision scalar variant

*Armv8-M Floating-point Extension only.*

Applies when **sz == 0**.

VRINTX{<c>}{<q>}.F32.F32 <Sd>, <Sm>

#### Double-precision scalar variant

*Armv8-M Floating-point Extension only.*

Applies when **sz == 1**.

VRINTX{<c>}{<q>}.F64.F64 <Dd>, <Dm>

#### Decode for this encoding

```

1 dp_operation = (sz == '1');
2 CheckDecodeFaults(dp_operation);
3 d = if dp_operation then UInt(D:Vd) else UInt(Vd:D);
4 m = if dp_operation then UInt(M:Vm) else UInt(Vm:M);

```

#### Assembler symbols for all encodings

<c> See [C1.2.5 Standard assembler syntax fields](#) on page 309.  
 <q> See [C1.2.5 Standard assembler syntax fields](#) on page 309.  
 <Sd> Is the 32-bit name of the floating-point destination register, encoded in the "Vd:D" field.  
 <Sm> Is the 32-bit name of the floating-point source register, encoded in the "Vm:M" field.  
 <Dd> Is the 64-bit name of the floating-point destination register, encoded in the "D:Vd" field.  
 <Dm> Is the 64-bit name of the floating-point source register, encoded in the "M:Vm" field.

#### Operation for all encodings

```

1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 ExecuteFPCheck();
4
5 rmode = FPSCR[23:22];
6 away = FALSE;
7 exact = TRUE;
8
9 if dp_operation then
10 D[d] = FPRoundInt(D[m], rmode, away, exact);
11 else
12 S[d] = FPRoundInt(S[m], rmode, away, exact);

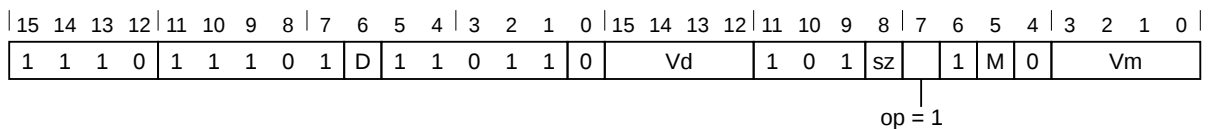
```

### C2.4.301 VRINTZ

Floating-point Round to Integer towards Zero. Floating-point Round to Integer towards Zero rounds a floating-point value to an integral floating-point value of the same size, using the Round towards Zero rounding mode. A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

#### T1

*Armv8-M Floating-point Extension only*, `sz == 1` UNDEFINED in single-precision only implementations.



#### Single-precision scalar variant

*Armv8-M Floating-point Extension only.*

Applies when `sz == 0`.

`VRINTZ{<c>}{<q>}.F32.F32 <Sd>, <Sm>`

#### Double-precision scalar variant

*Armv8-M Floating-point Extension only.*

Applies when `sz == 1`.

`VRINTZ{<c>}{<q>}.F64.F64 <Dd>, <Dm>`

#### Decode for this encoding

```

1 dp_operation = (sz == '1');
2 CheckDecodeFaults(dp_operation);
3 d = if dp_operation then UInt(D:Vd) else UInt(Vd:D);
4 m = if dp_operation then UInt(M:Vm) else UInt(Vm:M);

```

#### Assembler symbols for all encodings

- <c> See [C1.2.5 Standard assembler syntax fields](#) on page 309.
- <q> See [C1.2.5 Standard assembler syntax fields](#) on page 309.
- <Sd> Is the 32-bit name of the floating-point destination register, encoded in the "Vd:D" field.
- <Sm> Is the 32-bit name of the floating-point source register, encoded in the "Vm:M" field.
- <Dd> Is the 64-bit name of the floating-point destination register, encoded in the "D:Vd" field.
- <Dm> Is the 64-bit name of the floating-point source register, encoded in the "M:Vm" field.

#### Operation for all encodings

```

1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 ExecuteFPCheck();
4
5 rmode = if op == '1' then '11' else FPSCR[23:22];
6 exact = FALSE;
7 away = FALSE;
8
9 if dp_operation then
10 D[d] = FPRoundInt(D[m], rmode, away, exact);
11 else
12 S[d] = FPRoundInt(S[m], rmode, away, exact);

```

### C2.4.302 VSEL

Floating-point Conditional Select. Floating-point Conditional Select allows the destination register to take the value from either one or the other of two source registers according to the condition codes in the APSR.

The condition codes for VSEL are limited to GE, GT, EQ, and VS. The effect of LT, LE, NE, and VC can be achieved by exchanging the source operands.

#### T1

*Armv8-M Floating-point Extension only, sz == 1 UNDEFINED in single-precision only implementations.*

|    |    |    |    |    |    |   |   |   |   |    |    |    |   |   |   |    |    |    |    |    |    |   |   |   |   |   |   |   |   |   |   |
|----|----|----|----|----|----|---|---|---|---|----|----|----|---|---|---|----|----|----|----|----|----|---|---|---|---|---|---|---|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5  | 4  | 3  | 2 | 1 | 0 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 1  | 1  | 1  | 1  | 1  | 1  | 1 | 0 | 0 | D | cc | Vn | Vd | 1 | 0 | 1 | sz | N  | 0  | M  | 0  | Vm |   |   |   |   |   |   |   |   |   |   |

#### VSELEQ, Double-precision variant

*Armv8-M Floating-point Extension only.*

**Applies when cc == 00 && sz == 1.**

```
VSELEQ.F64 <Dd>, <Dn>, <Dm>
// Not permitted in IT block
```

#### VSELEQ, Single-precision variant

*Armv8-M Floating-point Extension only.*

**Applies when cc == 00 && sz == 0.**

```
VSELEQ.F32 <Sd>, <Sn>, <Sm>
// Not permitted in IT block
```

#### VSELGE, Double-precision variant

*Armv8-M Floating-point Extension only.*

**Applies when cc == 10 && sz == 1.**

```
VSELGE.F64 <Dd>, <Dn>, <Dm>
// Not permitted in IT block
```

#### VSELGE, Single-precision variant

*Armv8-M Floating-point Extension only.*

**Applies when cc == 10 && sz == 0.**

```
VSELGE.F32 <Sd>, <Sn>, <Sm>
// Not permitted in IT block
```

#### VSELGT, Double-precision variant

*Armv8-M Floating-point Extension only.*

**Applies when cc == 11 && sz == 1.**

```
VSELGT.F64 <Dd>, <Dn>, <Dm>
// Not permitted in IT block
```



### VSELGT, Single-precision variant

*Armv8-M Floating-point Extension only.*

**Applies when `cc == 11` && `sz == 0`.**

```
VSELGT.F32 <Sd>, <Sn>, <Sm>
 // Not permitted in IT block
```

### VSELVS, Double-precision variant

*Armv8-M Floating-point Extension only.*

**Applies when `cc == 01` && `sz == 1`.**

```
VSELVS.F64 <Dd>, <Dn>, <Dm>
 // Not permitted in IT block
```

### VSELVS, Single-precision variant

*Armv8-M Floating-point Extension only.*

**Applies when `cc == 01` && `sz == 0`.**

```
VSELVS.F32 <Sd>, <Sn>, <Sm>
 // Not permitted in IT block
```

### Decode for this encoding

```
1 dp_operation = (sz == '1');
2 CheckDecodeFaults(dp_operation);
3 if InITBlock() then UNPREDICTABLE;
4 cond = cc:(cc[1] EOR cc[0]):'0';
5 d = if dp_operation then UInt(D:Vd) else UInt(Vd:D);
6 n = if dp_operation then UInt(N:Vn) else UInt(Vn:N);
7 m = if dp_operation then UInt(M:Vm) else UInt(Vm:M);
```

### Assembler symbols for all encodings

|      |                                                                                               |
|------|-----------------------------------------------------------------------------------------------|
| <Dd> | Is the 64-bit name of the floating-point destination register, encoded in the "D:Vd" field.   |
| <Dn> | Is the 64-bit name of the first floating-point source register, encoded in the "N:Vn" field.  |
| <Dm> | Is the 64-bit name of the second floating-point source register, encoded in the "M:Vm" field. |
| <Sd> | Is the 32-bit name of the floating-point destination register, encoded in the "Vd:D" field.   |
| <Sn> | Is the 32-bit name of the first floating-point source register, encoded in the "Vn:N" field.  |
| <Sm> | Is the 32-bit name of the second floating-point source register, encoded in the "Vm:M" field. |

### Operation for all encodings

```
1 EncodingSpecificOperations();
2 ExecuteFPCheck();
3
4 if dp_operation then
5 D[d] = if ConditionHolds(cond) then D[n] else D[m];
6 else
7 S[d] = if ConditionHolds(cond) then S[n] else S[m];
```

### C2.4.303 VSQRT

Floating-point Square Root. Floating-point Square Root calculates the square root of a floating-point register value and writes the result to another floating-point register.

#### T1

*Armv8-M Floating-point Extension only*, *sz == 1 UNDEFINED* in single-precision only implementations.

|    |    |    |    |    |    |   |   |   |   |   |   |   |   |   |   |    |    |    |    |    |    |   |   |   |    |   |   |   |   |   |   |
|----|----|----|----|----|----|---|---|---|---|---|---|---|---|---|---|----|----|----|----|----|----|---|---|---|----|---|---|---|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6  | 5 | 4 | 3 | 2 | 1 | 0 |
| 1  | 1  | 1  | 0  | 1  | 1  | 1 | 0 | 1 | D | 1 | 1 | 0 | 0 | 0 | 1 | Vd | 1  | 0  | 1  | sz | 1  | 1 | M | 0 | Vm |   |   |   |   |   |   |

#### Single-precision scalar variant

*Armv8-M Floating-point Extension only*.

Applies when **sz == 0**.

VSQRT{<c>}{<q>}.F32 <Sd>, <Sm>

#### Double-precision scalar variant

*Armv8-M Floating-point Extension only*.

Applies when **sz == 1**.

VSQRT{<c>}{<q>}.F64 <Dd>, <Dm>

#### Decode for this encoding

```

1 dp_operation = (sz == '1');
2 CheckDecodeFaults(dp_operation);
3 d = if dp_operation then UInt(D:Vd) else UInt(Vd:D);
4 m = if dp_operation then UInt(M:Vm) else UInt(Vm:M);

```

#### Assembler symbols for all encodings

- <c> See [C1.2.5 Standard assembler syntax fields](#) on page 309.
- <q> See [C1.2.5 Standard assembler syntax fields](#) on page 309.
- <Sd> Is the 32-bit name of the floating-point destination register, encoded in the "Vd:D" field.
- <Sm> Is the 32-bit name of the floating-point source register, encoded in the "Vm:M" field.
- <Dd> Is the 64-bit name of the floating-point destination register, encoded in the "D:Vd" field.
- <Dm> Is the 64-bit name of the floating-point source register, encoded in the "M:Vm" field.

#### Operation for all encodings

```

1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 ExecuteFPCheck();
4 if dp_operation then
5 D[d] = FPSqrt(D[m]);
6 else
7 S[d] = FPSqrt(S[m]);

```

### C2.4.304 VSTM

Floating-point Store Multiple. Floating-point Store Multiple stores multiple extension registers to consecutive memory locations using an address from a general-purpose register.

This instruction is used by the alias [VPUSH](#).

#### T1

*Armv8-M Floating-point Extension only*

|    |    |    |    |    |    |   |   |   |   |   |   |    |   |   |   |    |    |    |    |    |    |   |   |             |   |   |   |   |   |   |   |
|----|----|----|----|----|----|---|---|---|---|---|---|----|---|---|---|----|----|----|----|----|----|---|---|-------------|---|---|---|---|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3  | 2 | 1 | 0 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7           | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 1  | 1  | 1  | 0  | 1  | 1  | 0 | P | U | D | W | 0 | Rn |   |   |   | Vd |    |    |    | 1  | 0  | 1 | 1 | imm8<0> = 0 |   |   |   |   |   |   |   |

#### Decrement Before variant

Applies when **P == 1 && U == 0 && W == 1**.

VSTMDB{<c>}{<q>}{.<size>} <Rn>!, <dreglist>

#### Increment After variant

Applies when **P == 0 && U == 1**.

VSTM{<c>}{<q>}{.<size>} <Rn>{!}, <dreglist>

VSTMIA{<c>}{<q>}{.<size>} <Rn>{!}, <dreglist>

#### Decode for this encoding

```

1 if P == '0' && U == '0' && W == '0' then SEE "Related encodings";
2 if P == '1' && W == '0' then SEE VSTR;
3 CheckDecodeFaults();
4 if P == U && W == '1' then UNDEFINED;
5 // Remaining combinations are PUW = 010 (IA without !), 011 (IA with !), 101 (DB with !)
6 single_regs = FALSE; add = (U == '1'); wback = (W == '1');
7 d = UInt(D:Vd); n = UInt(Rn); imm32 = ZeroExtend(imm8:'00', 32);
8 regs = UInt(imm8) DIV 2;
9 if n == 15 then UNPREDICTABLE;
10 if regs == 0 || regs > 16 || (d+regs) > 32 then UNPREDICTABLE;
11 if VFPSmallRegisterBank() && (d+regs) > 16 then UNPREDICTABLE;
```

#### CONSTRAINED UNPREDICTABLE behavior

If `regs == 0`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction operates as a [VSTM](#) with the same addressing mode but stores no registers.

#### CONSTRAINED UNPREDICTABLE behavior

If `(d+regs) > 32`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The memory locations specified by the instruction and the number of registers specified by the instruction if the register list had not gone out of range, become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

## T2

Armv8-M Floating-point Extension only

|    |    |    |    |    |    |   |   |   |   |   |   |    |   |   |   |    |    |    |    |         |    |   |   |      |   |   |   |   |   |   |   |
|----|----|----|----|----|----|---|---|---|---|---|---|----|---|---|---|----|----|----|----|---------|----|---|---|------|---|---|---|---|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3  | 2 | 1 | 0 | 15 | 14 | 13 | 12 | 11      | 10 | 9 | 8 | 7    | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 1  | 1  | 1  | 0  | 1  | 1  | 0 | P | U | D | W | 0 | Rn |   |   |   | Vd |    |    |    | 1 0 1 0 |    |   |   | imm8 |   |   |   |   |   |   |   |

### Decrement Before variant

Applies when **P == 1 && U == 0 && W == 1**.

VSTMDB{<c>}{<q>}{.<size>} <Rn>!, <sreglist>

### Increment After variant

Applies when **P == 0 && U == 1**.

VSTM{<c>}{<q>}{.<size>} <Rn>{!}, <sreglist>

VSTMIA{<c>}{<q>}{.<size>} <Rn>{!}, <sreglist>

### Decode for this encoding

```

1 if P == '0' && U == '0' then SEE "Related encodings";
2 if P == '1' && W == '0' then SEE VSTR;
3 CheckDecodeFaults();
4 if P == '1' && U == '1' && W == '1' then UNDEFINED;
5 // Remaining combinations are P UW = 010 (IA without !), 011 (IA with !), 101 (DB with !)
6 single_regs = TRUE; add = (U == '1'); wback = (W == '1'); d = UInt(Vd:D); n = UInt(Rn);
7 imm32 = ZeroExtend(imm8:'00', 32); regs = UInt(imm8);
8 if n == 15 then UNPREDICTABLE;
9 if regs == 0 || (d+regs) > 32 then UNPREDICTABLE;

```

### CONSTRAINED UNPREDICTABLE behavior

If `regs == 0`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction operates as a **VSTM** with the same addressing mode but stores no registers.

### CONSTRAINED UNPREDICTABLE behavior

If `(d+regs) > 64`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The memory locations specified by the instruction and the number of registers specified by the instruction if the register list had not gone out of range, become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

### Alias conditions

| Alias        | is preferred when                                                                     |
|--------------|---------------------------------------------------------------------------------------|
| <b>VPUSH</b> | <code>P == '1' &amp;&amp; U == '0' &amp;&amp; W == '1' &amp;&amp; RN == '1101'</code> |

### Assembler symbols for all encodings

<c> See [C1.2.5 Standard assembler syntax fields](#) on page 309.  
<q> See [C1.2.5 Standard assembler syntax fields](#) on page 309.

|            |                                                                                                                                                                                                                                                                                                         |
|------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <size>     | An optional data size specifier. If present, it must be equal to the size in bits, 32 or 64, of the registers being transferred.                                                                                                                                                                        |
| <Rn>       | Is the general-purpose base register, encoded in the "Rn" field.                                                                                                                                                                                                                                        |
| !          | Specifies base register writeback. Encoded in the "W" field as 1 if present, otherwise 0.                                                                                                                                                                                                               |
| <sreglist> | Is the list of consecutively numbered 32-bit floating-point registers to be transferred. The first register in the list is encoded in "Vd:D", and "imm8" is set to the number of registers in the list. The list must contain at least one register.                                                    |
| <dreglist> | Is the list of consecutively numbered 64-bit floating-point registers to be transferred. The first register in the list is encoded in "D:Vd", and "imm8" is set to twice the number of registers in the list. The list must contain at least one register, and must not contain more than 16 registers. |

### Operation for all encodings

```

1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 ExecuteFPCheck();
4 address = if add then R[n] else R[n]-imm32;
5 regval = if add then R[n]+imm32 else R[n]-imm32;
6
7 // Determine if the stack pointer limit should be checked
8 if n == 13 && wback then
9 (limit, applylimit) = LookUpSPLim(LookUpSP());
10 else
11 applylimit = FALSE;
12
13 // Memory operation only performed if limit not violated
14 if !applylimit || (UInt(regval) >= UInt(limit)) then
15 for r = 0 to regs-1
16 if single_regs then
17 MemA[address,4] = S[d+r];
18 address = address+4;
19 else
20 // Store as two word-aligned words in the correct order for current
21 // endianness.
22 MemA[address,4] = if BigEndian() then D[d+r][63:32] else D[d+r][31:0];
23 MemA[address+4,4] = if BigEndian() then D[d+r][31:0] else D[d+r][63:32];
24 address = address+8;
25
26 // If the stack pointer is being updated a fault will be raised if
27 // the limit is violated
28 if wback then RSPCheck[n] = regval;

```

### C2.4.305 VSTR

Floating-point Store Register. Floating-point Store Register stores a single Floating-point Extension register to memory, using an address from a general-purpose register, with an optional offset.

#### T1

Armv8-M Floating-point Extension only

|    |    |    |    |    |    |   |   |   |   |   |   |    |   |   |   |    |    |    |    |         |    |   |   |      |   |   |   |   |   |   |   |
|----|----|----|----|----|----|---|---|---|---|---|---|----|---|---|---|----|----|----|----|---------|----|---|---|------|---|---|---|---|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3  | 2 | 1 | 0 | 15 | 14 | 13 | 12 | 11      | 10 | 9 | 8 | 7    | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 1  | 1  | 1  | 0  | 1  | 1  | 0 | 1 | U | D | 0 | 0 | Rn |   |   |   | Vd |    |    |    | 1 0 1 1 |    |   |   | imm8 |   |   |   |   |   |   |   |

#### T1 variant

VSTR{<c>}{<q>}{.64} <Dd>, [<Rn>{, # {+/-}<imm>}]

#### Decode for this encoding

```

1 CheckDecodeFaults();
2 single_reg = FALSE; add = (U == '1'); imm32 = ZeroExtend(imm8:'00', 32);
3 d = UInt(D:Vd); n = UInt(Rn);
4 if n == 15 then UNPREDICTABLE;
```

#### T2

Armv8-M Floating-point Extension only

|    |    |    |    |    |    |   |   |   |   |   |   |    |   |   |   |    |    |    |    |         |    |   |   |      |   |   |   |   |   |   |   |
|----|----|----|----|----|----|---|---|---|---|---|---|----|---|---|---|----|----|----|----|---------|----|---|---|------|---|---|---|---|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3  | 2 | 1 | 0 | 15 | 14 | 13 | 12 | 11      | 10 | 9 | 8 | 7    | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 1  | 1  | 1  | 0  | 1  | 1  | 0 | 1 | U | D | 0 | 0 | Rn |   |   |   | Vd |    |    |    | 1 0 1 0 |    |   |   | imm8 |   |   |   |   |   |   |   |

#### T2 variant

VSTR{<c>}{<q>}{.32} <Sd>, [<Rn>{, # {+/-}<imm>}]

#### Decode for this encoding

```

1 CheckDecodeFaults();
2 single_reg = TRUE; add = (U == '1'); imm32 = ZeroExtend(imm8:'00', 32);
3 d = UInt(Vd:D); n = UInt(Rn);
4 if n == 15 then UNPREDICTABLE;
```

#### Assembler symbols for all encodings

|       |                                                                                                                                                                                                                                                        |
|-------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <c>   | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                                                                                                                                                                               |
| <q>   | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                                                                                                                                                                               |
| .64   | Optional data size specifiers.                                                                                                                                                                                                                         |
| <Dd>  | The source register for a doubleword store.                                                                                                                                                                                                            |
| .32   | Optional data size specifiers.                                                                                                                                                                                                                         |
| <Sd>  | The source register for a singleword store.                                                                                                                                                                                                            |
| <Rn>  | Is the general-purpose base register, encoded in the "Rn" field.                                                                                                                                                                                       |
| +/-   | Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in the "U" field. It can have the following values: <ul style="list-style-type: none"> <li>- when U = 0</li> <li>+ when U = 1</li> </ul> |
| <imm> | The immediate offset used for forming the address. Values are multiples of 4 in the range 0-1020. <imm> can be omitted, meaning an offset of +0.                                                                                                       |

**Operation for all encodings**

```
1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 ExecuteFPCheck();
4 address = if add then (R[n] + imm32) else (R[n] - imm32);
5 if single_reg then
6 MemA[address,4] = S[d];
7 else
8 // Store as two word-aligned words in the correct order for current endianness.
9 MemA[address,4] = if BigEndian() then D[d][63:32] else D[d][31:0];
10 MemA[address+4,4] = if BigEndian() then D[d][31:0] else D[d][63:32];
```

## C2.4.306 VSUB

Floating-point Subtract. Floating-point Subtract subtracts one floating-point register value from another floating-point register value, and places the results in the destination floating-point register.

### T2

Armv8-M Floating-point Extension only,  $sz == 1$  UNDEFINED in single-precision only implementations.

|    |    |    |    |    |    |   |   |   |   |   |   |    |   |   |   |    |    |    |    |    |    |   |    |   |   |   |   |    |   |   |   |
|----|----|----|----|----|----|---|---|---|---|---|---|----|---|---|---|----|----|----|----|----|----|---|----|---|---|---|---|----|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3  | 2 | 1 | 0 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8  | 7 | 6 | 5 | 4 | 3  | 2 | 1 | 0 |
| 1  | 1  | 1  | 0  | 1  | 1  | 1 | 0 | 0 | D | 1 | 1 | Vn |   |   |   | Vd |    |    |    | 1  | 0  | 1 | sz | N | 1 | M | 0 | Vm |   |   |   |

#### Single-precision scalar variant

Armv8-M Floating-point Extension only.

Applies when  $sz == 0$ .

`VSUB{<c>}{<q>}.F32 {<Sd>, } <Sn>, <Sm>`

#### Double-precision scalar variant

Armv8-M Floating-point Extension only.

Applies when  $sz == 1$ .

`VSUB{<c>}{<q>}.F64 {<Dd>, } <Dn>, <Dm>`

#### Decode for this encoding

```

1 dp_operation = (sz == '1');
2 CheckDecodeFaults(dp_operation);
3 d = if dp_operation then UInt(D:Vd) else UInt(Vd:D);
4 n = if dp_operation then UInt(N:Vn) else UInt(Vn:N);
5 m = if dp_operation then UInt(M:Vm) else UInt(Vm:M);

```

#### Assembler symbols for all encodings

|      |                                                                                               |
|------|-----------------------------------------------------------------------------------------------|
| <c>  | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                      |
| <q>  | See <a href="#">C1.2.5 Standard assembler syntax fields</a> on page 309.                      |
| <Sd> | Is the 32-bit name of the floating-point destination register, encoded in the "Vd:D" field.   |
| <Sn> | Is the 32-bit name of the first floating-point source register, encoded in the "Vn:N" field.  |
| <Sm> | Is the 32-bit name of the second floating-point source register, encoded in the "Vm:M" field. |
| <Dd> | Is the 64-bit name of the floating-point destination register, encoded in the "D:Vd" field.   |
| <Dn> | Is the 64-bit name of the first floating-point source register, encoded in the "N:Vn" field.  |
| <Dm> | Is the 64-bit name of the second floating-point source register, encoded in the "M:Vm" field. |

#### Operation for all encodings

```

1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 ExecuteFPCheck();
4 if dp_operation then
5 D[d] = FPSub(D[n], D[m], TRUE);
6 else
7 S[d] = FPSub(S[n], S[m], TRUE);

```



### C2.4.307 WFE

Wait For Event. Wait For Event is a hint instruction. If the Event Register is clear, it suspends execution in the lowest power state available consistent with a fast wakeup without the need for software restoration, until a reset, exception or other event occurs.

This is a NOP-compatible hint.

#### T1

Armv8-M

|    |    |    |    |    |    |   |   |   |   |   |   |   |   |   |   |
|----|----|----|----|----|----|---|---|---|---|---|---|---|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 1  | 0  | 1  | 1  | 1  | 1  | 1 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |

#### T1 variant

WFE{<c>}{<q>}

#### Decode for this encoding

```
1 // No additional decoding required
```

#### T2

Armv8-M Main Extension only

|    |    |    |    |    |    |   |   |   |   |   |   |     |     |     |     |    |    |     |    |     |    |   |   |   |   |   |   |   |   |   |   |
|----|----|----|----|----|----|---|---|---|---|---|---|-----|-----|-----|-----|----|----|-----|----|-----|----|---|---|---|---|---|---|---|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3   | 2   | 1   | 0   | 15 | 14 | 13  | 12 | 11  | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 1  | 1  | 1  | 1  | 0  | 0  | 1 | 1 | 1 | 0 | 1 | 0 | (1) | (1) | (1) | (1) | 1  | 0  | (0) | 0  | (0) | 0  | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |   |   |

#### T2 variant

WFE{<c>}.W

#### Decode for this encoding

```
1 if !HaveMainExt() then UNDEFINED;
2 // No additional decoding required
```

#### Assembler symbols for all encodings

<c> See [C1.2.5 Standard assembler syntax fields](#) on page 309.

<q> See [C1.2.5 Standard assembler syntax fields](#) on page 309.

#### Operation for all encodings

```
1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 if EventRegistered() then
4 ClearEventRegister();
5 else
6 WaitForEvent();
```

### C2.4.308 WFI

Wait For Interrupt. Wait For Interrupt is a hint instruction. It suspends execution, in the lowest power state available consistent with a fast wakeup without the need for software restoration, until a reset, asynchronous exception or other event occurs.

This is a NOP-compatible hint.

#### T1

Armv8-M

|    |    |    |    |    |    |   |   |   |   |   |   |   |   |   |   |
|----|----|----|----|----|----|---|---|---|---|---|---|---|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 1  | 0  | 1  | 1  | 1  | 1  | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 |

#### T1 variant

WFI{<c>}{<q>}

#### Decode for this encoding

```
1 // No additional decoding required
```

#### T2

Armv8-M Main Extension only

|    |    |    |    |    |    |   |   |   |   |   |   |     |     |     |     |    |    |     |    |     |    |   |   |   |   |   |   |   |   |   |   |
|----|----|----|----|----|----|---|---|---|---|---|---|-----|-----|-----|-----|----|----|-----|----|-----|----|---|---|---|---|---|---|---|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3   | 2   | 1   | 0   | 15 | 14 | 13  | 12 | 11  | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 1  | 1  | 1  | 1  | 0  | 0  | 1 | 1 | 1 | 0 | 1 | 0 | (1) | (1) | (1) | (1) | 1  | 0  | (0) | 0  | (0) | 0  | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 |

#### T2 variant

WFI{<c>}.W

#### Decode for this encoding

```
1 if !HaveMainExt() then UNDEFINED;
2 // No additional decoding required
```

#### Assembler symbols for all encodings

<c> See [C1.2.5 Standard assembler syntax fields](#) on page 309.  
 <q> See [C1.2.5 Standard assembler syntax fields](#) on page 309.

#### Operation for all encodings

```
1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 WaitForInterrupt();
```

### C2.4.309 YIELD

Yield hint. Yield is a hint instruction. It enables software with a multithreading capability to indicate to the hardware that it is performing a task, for example a spinlock, that could be swapped out to improve overall system performance. Hardware can use this hint to suspend and resume multiple code threads if it supports the capability.

This is a NOP-compatible hint.

#### T1

Armv8-M

|    |    |    |    |    |    |   |   |   |   |   |   |   |   |   |   |
|----|----|----|----|----|----|---|---|---|---|---|---|---|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 1  | 0  | 1  | 1  | 1  | 1  | 1 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |

#### T1 variant

YIELD{<c>}{<q>}

#### Decode for this encoding

```
1 // No additional decoding required
```

#### T2

Armv8-M Main Extension only

|    |    |    |    |    |    |   |   |   |   |   |   |     |     |     |     |    |    |     |    |     |    |   |   |   |   |   |   |   |   |   |   |
|----|----|----|----|----|----|---|---|---|---|---|---|-----|-----|-----|-----|----|----|-----|----|-----|----|---|---|---|---|---|---|---|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3   | 2   | 1   | 0   | 15 | 14 | 13  | 12 | 11  | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 1  | 1  | 1  | 1  | 0  | 0  | 1 | 1 | 1 | 0 | 1 | 0 | (1) | (1) | (1) | (1) | 1  | 0  | (0) | 0  | (0) | 0  | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |   |   |

#### T2 variant

YIELD{<c>}.W

#### Decode for this encoding

```
1 if !HaveMainExt() then UNDEFINED;
2 // No additional decoding required
```

#### Assembler symbols for all encodings

<c> See [C1.2.5 Standard assembler syntax fields](#) on page 309.

<q> See [C1.2.5 Standard assembler syntax fields](#) on page 309.

#### Operation for all encodings

```
1 if ConditionPassed() then
2 EncodingSpecificOperations();
3 Hint_Yield();
```

**Part D**  
**Armv8-M Registers**

## Chapter D1

# Register Specification

This chapter specifies the Armv8-M registers. It contains the following sections:

[Register Index](#)

[Alphabetical list of registers](#)

## D1.1 Register index

| Address    | Component                                       |
|------------|-------------------------------------------------|
| –          | Special and general-purpose registers           |
| –          | Payloads                                        |
| 0xE0000000 | Instrumentation Macrocell                       |
| 0xE0001000 | Data Watchpoint and Trace                       |
| 0xE0002000 | Flash Patch and Breakpoint                      |
| 0xE000E004 | Implementation Control Block                    |
| 0xE000E010 | SysTick Timer                                   |
| 0xE000E100 | Nested Vectored Interrupt Controller            |
| 0xE000ED00 | System Control Block                            |
| 0xE000ED90 | Memory Protection Unit                          |
| 0xE000EDD0 | Security Attribution Unit                       |
| 0xE000EDF0 | Debug Control Block                             |
| 0xE000EF00 | Software Interrupt Generation                   |
| 0xE000EF34 | Floating-Point Extension                        |
| 0xE000EF50 | Cache Maintenance Operations                    |
| 0xE000EFB0 | Debug Identification Block                      |
| 0xE002E004 | Implementation Control Block (NS alias)         |
| 0xE002E010 | SysTick Timer (NS alias)                        |
| 0xE002E100 | Nested Vectored Interrupt Controller (NS alias) |
| 0xE002ED00 | System Control Block (NS alias)                 |
| 0xE002ED90 | Memory Protection Unit (NS alias)               |
| 0xE002EDF0 | Debug Control Block (NS alias)                  |
| 0xE002EF00 | Software Interrupt Generation (NS alias)        |
| 0xE002EF34 | Floating-Point Extension (NS alias)             |
| 0xE002EF50 | Cache Maintenance Operations (NS alias)         |
| 0xE002EFB0 | Debug Identification Block (NS alias)           |
| 0xE0040000 | Trace Port Interface Unit                       |

### D1.1.1 Special and general-purpose registers

| Name      | Description                                |
|-----------|--------------------------------------------|
| APSR      | Application Program Status Register        |
| BASEPRI   | Base Priority Mask Register                |
| CONTROL   | Control Register                           |
| EPSR      | Execution Program Status Register          |
| FAULTMASK | Fault Mask Register                        |
| FPSCR     | Floating-point Status and Control Register |
| IPSR      | Interrupt Program Status Register          |
| LR        | Link Register                              |
| MSPLIM    | Main Stack Pointer Limit Register          |
| PC        | Program Counter                            |
| PRIMASK   | Exception Mask Register                    |
| PSPLIM    | Process Stack Pointer Limit Register       |
| Rn        | General-Purpose Register <i>n</i>          |
| SP        | Current Stack Pointer Register             |
| SP        | Stack Pointer (Non-secure)                 |
| XPSR      | Combined Program Status Registers          |

### D1.1.2 Payloads

| Name                       | Description                                        |
|----------------------------|----------------------------------------------------|
| <a href="#">EXC_RETURN</a> | Exception Return Payload                           |
| <a href="#">FNC_RETURN</a> | Function Return Payload                            |
| <a href="#">MAIR_ATTR</a>  | Memory Attribute Indirection Register Attributes   |
| <a href="#">RETPSR</a>     | Combined Exception Return Program Status Registers |
| <a href="#">TT_RESP</a>    | Test Target Response Payload                       |

### D1.1.3 Instrumentation Macrocell

| Address    | Register                    | Description                              |
|------------|-----------------------------|------------------------------------------|
| 0xE0000000 | <a href="#">ITM_STIMn</a>   | ITM Stimulus Port Register <i>n</i>      |
| 0xE0000E00 | <a href="#">ITM_TERn</a>    | ITM Trace Enable Register <i>n</i>       |
| 0xE0000E40 | <a href="#">ITM_TPR</a>     | ITM Trace Privilege Register             |
| 0xE0000E80 | <a href="#">ITM_TCR</a>     | ITM Trace Control Register               |
| 0xE0000FB0 | <a href="#">ITM_LAR</a>     | ITM Software Lock Access Register        |
| 0xE0000FB4 | <a href="#">ITM_LSR</a>     | ITM Software Lock Status Register        |
| 0xE0000FBC | <a href="#">ITM_DEVARCH</a> | ITM Device Architecture Register         |
| 0xE0000FCC | <a href="#">ITM_DEVTYPE</a> | ITM Device Type Register                 |
| 0xE0000FD0 | <a href="#">ITM_PIDR4</a>   | ITM Peripheral Identification Register 4 |
| 0xE0000FD4 | <a href="#">ITM_PIDR5</a>   | ITM Peripheral Identification Register 5 |
| 0xE0000FD8 | <a href="#">ITM_PIDR6</a>   | ITM Peripheral Identification Register 6 |
| 0xE0000FDC | <a href="#">ITM_PIDR7</a>   | ITM Peripheral Identification Register 7 |
| 0xE0000FE0 | <a href="#">ITM_PIDR0</a>   | ITM Peripheral Identification Register 0 |
| 0xE0000FE4 | <a href="#">ITM_PIDR1</a>   | ITM Peripheral Identification Register 1 |
| 0xE0000FE8 | <a href="#">ITM_PIDR2</a>   | ITM Peripheral Identification Register 2 |
| 0xE0000FEC | <a href="#">ITM_PIDR3</a>   | ITM Peripheral Identification Register 3 |
| 0xE0000FF0 | <a href="#">ITM_CIDR0</a>   | ITM Component Identification Register 0  |
| 0xE0000FF4 | <a href="#">ITM_CIDR1</a>   | ITM Component Identification Register 1  |
| 0xE0000FF8 | <a href="#">ITM_CIDR2</a>   | ITM Component Identification Register 2  |
| 0xE0000FFC | <a href="#">ITM_CIDR3</a>   | ITM Component Identification Register 3  |

### D1.1.4 Data Watchpoint and Trace

| Address    | Register                      | Description                               |
|------------|-------------------------------|-------------------------------------------|
| 0xE0001000 | <a href="#">DWT_CTRL</a>      | DWT Control Register                      |
| 0xE0001004 | <a href="#">DWT_CYCCNT</a>    | DWT Cycle Count Register                  |
| 0xE0001008 | <a href="#">DWT_CPICNT</a>    | DWT CPI Count Register                    |
| 0xE000100C | <a href="#">DWT_EXCCNT</a>    | DWT Exception Overhead Count Register     |
| 0xE0001010 | <a href="#">DWT_SLEEPCNT</a>  | DWT Sleep Count Register                  |
| 0xE0001014 | <a href="#">DWT_LSUCNT</a>    | DWT LSU Count Register                    |
| 0xE0001018 | <a href="#">DWT_FOLDCNT</a>   | DWT Folded Instruction Count Register     |
| 0xE000101C | <a href="#">DWT_PCSR</a>      | DWT Program Counter Sample Register       |
| 0xE0001020 | <a href="#">DWT_COMPn</a>     | DWT Comparator Register <i>n</i>          |
| 0xE0001028 | <a href="#">DWT_FUNCTIONn</a> | DWT Comparator Function Register <i>n</i> |
| 0xE0001FB0 | <a href="#">DWT_LAR</a>       | DWT Software Lock Access Register         |
| 0xE0001FB4 | <a href="#">DWT_LSR</a>       | DWT Software Lock Status Register         |
| 0xE0001FBC | <a href="#">DWT_DEVARCH</a>   | DWT Device Architecture Register          |

| Address    | Register                    | Description                              |
|------------|-----------------------------|------------------------------------------|
| 0xE0001FCC | <a href="#">DWT_DEVTYPE</a> | DWT Device Type Register                 |
| 0xE0001FD0 | <a href="#">DWT_PIDR4</a>   | DWT Peripheral Identification Register 4 |
| 0xE0001FD4 | <a href="#">DWT_PIDR5</a>   | DWT Peripheral Identification Register 5 |
| 0xE0001FD8 | <a href="#">DWT_PIDR6</a>   | DWT Peripheral Identification Register 6 |
| 0xE0001FDC | <a href="#">DWT_PIDR7</a>   | DWT Peripheral Identification Register 7 |
| 0xE0001FE0 | <a href="#">DWT_PIDR0</a>   | DWT Peripheral Identification Register 0 |
| 0xE0001FE4 | <a href="#">DWT_PIDR1</a>   | DWT Peripheral Identification Register 1 |
| 0xE0001FE8 | <a href="#">DWT_PIDR2</a>   | DWT Peripheral Identification Register 2 |
| 0xE0001FEC | <a href="#">DWT_PIDR3</a>   | DWT Peripheral Identification Register 3 |
| 0xE0001FF0 | <a href="#">DWT_CIDR0</a>   | DWT Component Identification Register 0  |
| 0xE0001FF4 | <a href="#">DWT_CIDR1</a>   | DWT Component Identification Register 1  |
| 0xE0001FF8 | <a href="#">DWT_CIDR2</a>   | DWT Component Identification Register 2  |
| 0xE0001FFC | <a href="#">DWT_CIDR3</a>   | DWT Component Identification Register 3  |

### D1.1.5 Flash Patch and Breakpoint

| Address    | Register                   | Description                              |
|------------|----------------------------|------------------------------------------|
| 0xE0002000 | <a href="#">FP_CTRL</a>    | Flash Patch Control Register             |
| 0xE0002004 | <a href="#">FP_REMAP</a>   | Flash Patch Remap Register               |
| 0xE0002008 | <a href="#">FP_COMPn</a>   | Flash Patch Comparator Register <i>n</i> |
| 0xE0002FB0 | <a href="#">FP_LAR</a>     | FPB Software Lock Access Register        |
| 0xE0002FB4 | <a href="#">FP_LSR</a>     | FPB Software Lock Status Register        |
| 0xE0002FBC | <a href="#">FP_DEVARCH</a> | FPB Device Architecture Register         |
| 0xE0002FCC | <a href="#">FP_DEVTYPE</a> | FPB Device Type Register                 |
| 0xE0002FD0 | <a href="#">FP_PIDR4</a>   | FP Peripheral Identification Register 4  |
| 0xE0002FD4 | <a href="#">FP_PIDR5</a>   | FP Peripheral Identification Register 5  |
| 0xE0002FD8 | <a href="#">FP_PIDR6</a>   | FP Peripheral Identification Register 6  |
| 0xE0002FDC | <a href="#">FP_PIDR7</a>   | FP Peripheral Identification Register 7  |
| 0xE0002FE0 | <a href="#">FP_PIDR0</a>   | FP Peripheral Identification Register 0  |
| 0xE0002FE4 | <a href="#">FP_PIDR1</a>   | FP Peripheral Identification Register 1  |
| 0xE0002FE8 | <a href="#">FP_PIDR2</a>   | FP Peripheral Identification Register 2  |
| 0xE0002FEC | <a href="#">FP_PIDR3</a>   | FP Peripheral Identification Register 3  |
| 0xE0002FF0 | <a href="#">FP_CIDR0</a>   | FP Component Identification Register 0   |
| 0xE0002FF4 | <a href="#">FP_CIDR1</a>   | FP Component Identification Register 1   |
| 0xE0002FF8 | <a href="#">FP_CIDR2</a>   | FP Component Identification Register 2   |
| 0xE0002FFC | <a href="#">FP_CIDR3</a>   | FP Component Identification Register 3   |

### D1.1.6 Implementation Control Block

| Address    | Register              | Description                        |
|------------|-----------------------|------------------------------------|
| 0xE000E004 | <a href="#">ICTR</a>  | Interrupt Controller Type Register |
| 0xE000E008 | <a href="#">ACTLR</a> | Auxiliary Control Register         |
| 0xE000E00C | <a href="#">CPPWR</a> | Coprocessor Power Control Register |

### D1.1.7 SysTick Timer



| Address    | Register                   | Description                         |
|------------|----------------------------|-------------------------------------|
| 0xE000E010 | <a href="#">SYST_CSR</a>   | SysTick Control and Status Register |
| 0xE000E014 | <a href="#">SYST_RVR</a>   | SysTick Reload Value Register       |
| 0xE000E018 | <a href="#">SYST_CVR</a>   | SysTick Current Value Register      |
| 0xE000E01C | <a href="#">SYST_CALIB</a> | SysTick Calibration Value Register  |

### D1.1.8 Nested Vectored Interrupt Controller

| Address    | Register                   | Description                                   |
|------------|----------------------------|-----------------------------------------------|
| 0xE000E100 | <a href="#">NVIC_ISERn</a> | Interrupt Set Enable Register <i>n</i>        |
| 0xE000E180 | <a href="#">NVIC_ICERn</a> | Interrupt Clear Enable Register <i>n</i>      |
| 0xE000E200 | <a href="#">NVIC_ISPRn</a> | Interrupt Set Pending Register <i>n</i>       |
| 0xE000E280 | <a href="#">NVIC_ICPRn</a> | Interrupt Clear Pending Register <i>n</i>     |
| 0xE000E300 | <a href="#">NVIC_IABRn</a> | Interrupt Active Bit Register <i>n</i>        |
| 0xE000E380 | <a href="#">NVIC_ITNSn</a> | Interrupt Target Non-secure Register <i>n</i> |
| 0xE000E400 | <a href="#">NVIC_IPRn</a>  | Interrupt Priority Register <i>n</i>          |

### D1.1.9 System Control Block

| Address    | Register                 | Description                                      |
|------------|--------------------------|--------------------------------------------------|
| 0xE000ED00 | <a href="#">CPUID</a>    | CPUID Base Register                              |
| 0xE000ED04 | <a href="#">ICSR</a>     | Interrupt Control and State Register             |
| 0xE000ED08 | <a href="#">VTOR</a>     | Vector Table Offset Register                     |
| 0xE000ED0C | <a href="#">AIRCR</a>    | Application Interrupt and Reset Control Register |
| 0xE000ED10 | <a href="#">SCR</a>      | System Control Register                          |
| 0xE000ED14 | <a href="#">CCR</a>      | Configuration and Control Register               |
| 0xE000ED18 | <a href="#">SHPR1</a>    | System Handler Priority Register 1               |
| 0xE000ED1C | <a href="#">SHPR2</a>    | System Handler Priority Register 2               |
| 0xE000ED20 | <a href="#">SHPR3</a>    | System Handler Priority Register 3               |
| 0xE000ED24 | <a href="#">SHCSR</a>    | System Handler Control and State Register        |
| 0xE000ED28 | <a href="#">MMFSR</a>    | MemManage Fault Status Register                  |
| 0xE000ED28 | <a href="#">CFSR</a>     | Configurable Fault Status Register               |
| 0xE000ED29 | <a href="#">BFSR</a>     | BusFault Status Register                         |
| 0xE000ED2A | <a href="#">UFSR</a>     | UsageFault Status Register                       |
| 0xE000ED2C | <a href="#">HFSR</a>     | HardFault Status Register                        |
| 0xE000ED30 | <a href="#">DFSR</a>     | Debug Fault Status Register                      |
| 0xE000ED34 | <a href="#">MMFAR</a>    | MemManage Fault Address Register                 |
| 0xE000ED38 | <a href="#">BFAR</a>     | BusFault Address Register                        |
| 0xE000ED3C | <a href="#">AFSR</a>     | Auxiliary Fault Status Register                  |
| 0xE000ED40 | <a href="#">ID_PFR0</a>  | Processor Feature Register 0                     |
| 0xE000ED44 | <a href="#">ID_PFR1</a>  | Processor Feature Register 1                     |
| 0xE000ED48 | <a href="#">ID_DFR0</a>  | Debug Feature Register 0                         |
| 0xE000ED4C | <a href="#">ID_AFR0</a>  | Auxiliary Feature Register 0                     |
| 0xE000ED50 | <a href="#">ID_MMFR0</a> | Memory Model Feature Register 0                  |
| 0xE000ED54 | <a href="#">ID_MMFR1</a> | Memory Model Feature Register 1                  |
| 0xE000ED58 | <a href="#">ID_MMFR2</a> | Memory Model Feature Register 2                  |
| 0xE000ED5C | <a href="#">ID_MMFR3</a> | Memory Model Feature Register 3                  |
| 0xE000ED60 | <a href="#">ID_ISAR0</a> | Instruction Set Attribute Register 0             |
| 0xE000ED64 | <a href="#">ID_ISAR1</a> | Instruction Set Attribute Register 1             |

| Address    | Register                 | Description                          |
|------------|--------------------------|--------------------------------------|
| 0xE000ED68 | <a href="#">ID_ISAR2</a> | Instruction Set Attribute Register 2 |
| 0xE000ED6C | <a href="#">ID_ISAR3</a> | Instruction Set Attribute Register 3 |
| 0xE000ED70 | <a href="#">ID_ISAR4</a> | Instruction Set Attribute Register 4 |
| 0xE000ED74 | <a href="#">ID_ISAR5</a> | Instruction Set Attribute Register 5 |
| 0xE000ED78 | <a href="#">CLIDR</a>    | Cache Level ID Register              |
| 0xE000ED7C | <a href="#">CTR</a>      | Cache Type Register                  |
| 0xE000ED80 | <a href="#">CCSIDR</a>   | Current Cache Size ID register       |
| 0xE000ED84 | <a href="#">CSSELR</a>   | Cache Size Selection Register        |
| 0xE000ED88 | <a href="#">CPACR</a>    | Coprocessor Access Control Register  |
| 0xE000ED8C | <a href="#">NSACR</a>    | Non-secure Access Control Register   |

### D1.1.10 Memory Protection Unit

| Address    | Register                    | Description                                      |
|------------|-----------------------------|--------------------------------------------------|
| 0xE000ED90 | <a href="#">MPU_TYPE</a>    | MPU Type Register                                |
| 0xE000ED94 | <a href="#">MPU_CTRL</a>    | MPU Control Register                             |
| 0xE000ED98 | <a href="#">MPU_RNR</a>     | MPU Region Number Register                       |
| 0xE000ED9C | <a href="#">MPU_RBAR</a>    | MPU Region Base Address Register                 |
| 0xE000EDA0 | <a href="#">MPU_RLAR</a>    | MPU Region Limit Address Register                |
| 0xE000EDA4 | <a href="#">MPU_RBAR_An</a> | MPU Region Base Address Register Alias <i>n</i>  |
| 0xE000EDA8 | <a href="#">MPU_RLAR_An</a> | MPU Region Limit Address Register Alias <i>n</i> |
| 0xE000EDC0 | <a href="#">MPU_MAIRO</a>   | MPU Memory Attribute Indirection Register 0      |
| 0xE000EDC4 | <a href="#">MPU_MAIR1</a>   | MPU Memory Attribute Indirection Register 1      |

### D1.1.11 Security Attribution Unit

| Address    | Register                 | Description                       |
|------------|--------------------------|-----------------------------------|
| 0xE000EDD0 | <a href="#">SAU_CTRL</a> | SAU Control Register              |
| 0xE000EDD4 | <a href="#">SAU_TYPE</a> | SAU Type Register                 |
| 0xE000EDD8 | <a href="#">SAU_RNR</a>  | SAU Region Number Register        |
| 0xE000EDDC | <a href="#">SAU_RBAR</a> | SAU Region Base Address Register  |
| 0xE000EDE0 | <a href="#">SAU_RLAR</a> | SAU Region Limit Address Register |
| 0xE000EDE4 | <a href="#">SFSR</a>     | Secure Fault Status Register      |
| 0xE000EDE8 | <a href="#">SFAR</a>     | Secure Fault Address Register     |

### D1.1.12 Debug Control Block

| Address    | Register                  | Description                                  |
|------------|---------------------------|----------------------------------------------|
| 0xE000EDF0 | <a href="#">DHCSR</a>     | Debug Halting Control and Status Register    |
| 0xE000EDF4 | <a href="#">DCRSR</a>     | Debug Core Register Select Register          |
| 0xE000EDF8 | <a href="#">DCRDR</a>     | Debug Core Register Data Register            |
| 0xE000EDFC | <a href="#">DEMCR</a>     | Debug Exception and Monitor Control Register |
| 0xE000EE04 | <a href="#">DAUTHCTRL</a> | Debug Authentication Control Register        |
| 0xE000EE08 | <a href="#">DSCSR</a>     | Debug Security Control and Status Register   |

**D1.1.13 Software Interrupt Generation**

| Address    | Register             | Description                           |
|------------|----------------------|---------------------------------------|
| 0xE000EF00 | <a href="#">STIR</a> | Software Triggered Interrupt Register |

**D1.1.14 Floating-Point Extension**

| Address    | Register               | Description                                    |
|------------|------------------------|------------------------------------------------|
| 0xE000EF34 | <a href="#">FPCCR</a>  | Floating-Point Context Control Register        |
| 0xE000EF38 | <a href="#">FPCAR</a>  | Floating-Point Context Address Register        |
| 0xE000EF3C | <a href="#">FPDSCR</a> | Floating-Point Default Status Control Register |
| 0xE000EF40 | <a href="#">MVFRO</a>  | Media and VFP Feature Register 0               |
| 0xE000EF44 | <a href="#">MVFR1</a>  | Media and VFP Feature Register 1               |
| 0xE000EF48 | <a href="#">MVFR2</a>  | Media and VFP Feature Register 2               |

**D1.1.15 Cache Maintenance Operations**

| Address    | Register                 | Description                                            |
|------------|--------------------------|--------------------------------------------------------|
| 0xE000EF50 | <a href="#">ICIALLU</a>  | Instruction Cache Invalidate All to PoU                |
| 0xE000EF58 | <a href="#">ICIMVAU</a>  | Instruction Cache line Invalidate by Address to PoU    |
| 0xE000EF5C | <a href="#">DCIMVAC</a>  | Data Cache line Invalidate by Address to PoC           |
| 0xE000EF60 | <a href="#">DCISW</a>    | Data Cache line Invalidate by Set/Way                  |
| 0xE000EF64 | <a href="#">DCCMVAU</a>  | Data Cache line Clean by address to PoU                |
| 0xE000EF68 | <a href="#">DCCMVAC</a>  | Data Cache line Clean by Address to PoC                |
| 0xE000EF6C | <a href="#">DCCSW</a>    | Data Cache Clean line by Set/Way                       |
| 0xE000EF70 | <a href="#">DCCIMVAC</a> | Data Cache line Clean and Invalidate by Address to PoC |
| 0xE000EF74 | <a href="#">DCCISW</a>   | Data Cache line Clean and Invalidate by Set/Way        |
| 0xE000EF78 | <a href="#">BPIALL</a>   | Branch Predictor Invalidate All                        |

**D1.1.16 Debug Identification Block**

| Address    | Register                    | Description                              |
|------------|-----------------------------|------------------------------------------|
| 0xE000EFB0 | <a href="#">DLAR</a>        | SCS Software Lock Access Register        |
| 0xE000EFB4 | <a href="#">DLSR</a>        | SCS Software Lock Status Register        |
| 0xE000EFB8 | <a href="#">DAUTHSTATUS</a> | Debug Authentication Status Register     |
| 0xE000EFBC | <a href="#">DDEVARCH</a>    | SCS Device Architecture Register         |
| 0xE000EFCC | <a href="#">DDEVTYPE</a>    | SCS Device Type Register                 |
| 0xE000EFD0 | <a href="#">DPIDR4</a>      | SCS Peripheral Identification Register 4 |
| 0xE000EFD4 | <a href="#">DPIDR5</a>      | SCS Peripheral Identification Register 5 |
| 0xE000EFD8 | <a href="#">DPIDR6</a>      | SCS Peripheral Identification Register 6 |
| 0xE000EFD8 | <a href="#">DPIDR7</a>      | SCS Peripheral Identification Register 7 |
| 0xE000EFE0 | <a href="#">DPIDR0</a>      | SCS Peripheral Identification Register 0 |
| 0xE000EFE4 | <a href="#">DPIDR1</a>      | SCS Peripheral Identification Register 1 |
| 0xE000EFE8 | <a href="#">DPIDR2</a>      | SCS Peripheral Identification Register 2 |
| 0xE000EFEC | <a href="#">DPIDR3</a>      | SCS Peripheral Identification Register 3 |
| 0xE000EFF0 | <a href="#">DCIDR0</a>      | SCS Component Identification Register 0  |

| Address    | Register               | Description                             |
|------------|------------------------|-----------------------------------------|
| 0xE000EFF4 | <a href="#">DCIDR1</a> | SCS Component Identification Register 1 |
| 0xE000EFF8 | <a href="#">DCIDR2</a> | SCS Component Identification Register 2 |
| 0xE000EFFC | <a href="#">DCIDR3</a> | SCS Component Identification Register 3 |

### D1.1.17 Implementation Control Block (NS alias)

| Address    | Register              | Description                             |
|------------|-----------------------|-----------------------------------------|
| 0xE002E004 | <a href="#">ICTR</a>  | Interrupt Controller Type Register (NS) |
| 0xE002E008 | <a href="#">ACTLR</a> | Auxiliary Control Register (NS)         |
| 0xE002E00C | <a href="#">CPPWR</a> | Coprocessor Power Control Register (NS) |

### D1.1.18 SysTick Timer (NS alias)

| Address    | Register                   | Description                              |
|------------|----------------------------|------------------------------------------|
| 0xE002E010 | <a href="#">SYST_CSR</a>   | SysTick Control and Status Register (NS) |
| 0xE002E014 | <a href="#">SYST_RVR</a>   | SysTick Reload Value Register (NS)       |
| 0xE002E018 | <a href="#">SYST_CVR</a>   | SysTick Current Value Register (NS)      |
| 0xE002E01C | <a href="#">SYST_CALIB</a> | SysTick Calibration Value Register (NS)  |

### D1.1.19 Nested Vectored Interrupt Controller (NS alias)

| Address    | Register                   | Description                                    |
|------------|----------------------------|------------------------------------------------|
| 0xE002E100 | <a href="#">NVIC_ISERn</a> | Interrupt Set Enable Register <i>n</i> (NS)    |
| 0xE002E180 | <a href="#">NVIC_ICERn</a> | Interrupt Clear Enable Register <i>n</i> (NS)  |
| 0xE002E200 | <a href="#">NVIC_ISPRn</a> | Interrupt Set Pending Register <i>n</i> (NS)   |
| 0xE002E280 | <a href="#">NVIC_ICPRn</a> | Interrupt Clear Pending Register <i>n</i> (NS) |
| 0xE002E300 | <a href="#">NVIC_IABRn</a> | Interrupt Active Bit Register <i>n</i> (NS)    |
| 0xE002E400 | <a href="#">NVIC_IPRn</a>  | Interrupt Priority Register <i>n</i> (NS)      |

### D1.1.20 System Control Block (NS alias)

| Address    | Register              | Description                                           |
|------------|-----------------------|-------------------------------------------------------|
| 0xE002ED00 | <a href="#">CPUID</a> | CPUID Base Register (NS)                              |
| 0xE002ED04 | <a href="#">ICSR</a>  | Interrupt Control and State Register (NS)             |
| 0xE002ED08 | <a href="#">VTOR</a>  | Vector Table Offset Register (NS)                     |
| 0xE002ED0C | <a href="#">AIRCR</a> | Application Interrupt and Reset Control Register (NS) |
| 0xE002ED10 | <a href="#">SCR</a>   | System Control Register (NS)                          |
| 0xE002ED14 | <a href="#">CCR</a>   | Configuration and Control Register (NS)               |
| 0xE002ED18 | <a href="#">SHPR1</a> | System Handler Priority Register 1 (NS)               |
| 0xE002ED1C | <a href="#">SHPR2</a> | System Handler Priority Register 2 (NS)               |
| 0xE002ED20 | <a href="#">SHPR3</a> | System Handler Priority Register 3 (NS)               |
| 0xE002ED24 | <a href="#">SHCSR</a> | System Handler Control and State Register (NS)        |
| 0xE002ED28 | <a href="#">CFSR</a>  | Configurable Fault Status Register (NS)               |

| Address    | Register                 | Description                               |
|------------|--------------------------|-------------------------------------------|
| 0xE002ED28 | <a href="#">MMFSR</a>    | MemManage Fault Status Register (NS)      |
| 0xE002ED29 | <a href="#">BFSR</a>     | BusFault Status Register (NS)             |
| 0xE002ED2A | <a href="#">UFSR</a>     | UsageFault Status Register (NS)           |
| 0xE002ED2C | <a href="#">HFSR</a>     | HardFault Status Register (NS)            |
| 0xE002ED30 | <a href="#">DFSR</a>     | Debug Fault Status Register (NS)          |
| 0xE002ED34 | <a href="#">MMFAR</a>    | MemManage Fault Address Register (NS)     |
| 0xE002ED38 | <a href="#">BFAR</a>     | BusFault Address Register (NS)            |
| 0xE002ED3C | <a href="#">AFSR</a>     | Auxiliary Fault Status Register (NS)      |
| 0xE002ED40 | <a href="#">ID_PFR0</a>  | Processor Feature Register 0 (NS)         |
| 0xE002ED44 | <a href="#">ID_PFR1</a>  | Processor Feature Register 1 (NS)         |
| 0xE002ED48 | <a href="#">ID_DFR0</a>  | Debug Feature Register 0 (NS)             |
| 0xE002ED4C | <a href="#">ID_AFR0</a>  | Auxiliary Feature Register 0 (NS)         |
| 0xE002ED50 | <a href="#">ID_MMFR0</a> | Memory Model Feature Register 0 (NS)      |
| 0xE002ED54 | <a href="#">ID_MMFR1</a> | Memory Model Feature Register 1 (NS)      |
| 0xE002ED58 | <a href="#">ID_MMFR2</a> | Memory Model Feature Register 2 (NS)      |
| 0xE002ED5C | <a href="#">ID_MMFR3</a> | Memory Model Feature Register 3 (NS)      |
| 0xE002ED60 | <a href="#">ID_ISAR0</a> | Instruction Set Attribute Register 0 (NS) |
| 0xE002ED64 | <a href="#">ID_ISAR1</a> | Instruction Set Attribute Register 1 (NS) |
| 0xE002ED68 | <a href="#">ID_ISAR2</a> | Instruction Set Attribute Register 2 (NS) |
| 0xE002ED6C | <a href="#">ID_ISAR3</a> | Instruction Set Attribute Register 3 (NS) |
| 0xE002ED70 | <a href="#">ID_ISAR4</a> | Instruction Set Attribute Register 4 (NS) |
| 0xE002ED74 | <a href="#">ID_ISAR5</a> | Instruction Set Attribute Register 5 (NS) |
| 0xE002ED78 | <a href="#">CLIDR</a>    | Cache Level ID Register (NS)              |
| 0xE002ED7C | <a href="#">CTR</a>      | Cache Type Register (NS)                  |
| 0xE002ED80 | <a href="#">CCSIDR</a>   | Current Cache Size ID register (NS)       |
| 0xE002ED84 | <a href="#">CSSELR</a>   | Cache Size Selection Register (NS)        |
| 0xE002ED88 | <a href="#">CPACR</a>    | Coprocessor Access Control Register (NS)  |

### D1.1.21 Memory Protection Unit (NS alias)

| Address    | Register                    | Description                                           |
|------------|-----------------------------|-------------------------------------------------------|
| 0xE002ED90 | <a href="#">MPU_TYPE</a>    | MPU Type Register (NS)                                |
| 0xE002ED94 | <a href="#">MPU_CTRL</a>    | MPU Control Register (NS)                             |
| 0xE002ED98 | <a href="#">MPU_RNR</a>     | MPU Region Number Register (NS)                       |
| 0xE002ED9C | <a href="#">MPU_RBAR</a>    | MPU Region Base Address Register (NS)                 |
| 0xE002EDA0 | <a href="#">MPU_RLAR</a>    | MPU Region Limit Address Register (NS)                |
| 0xE002EDA4 | <a href="#">MPU_RBAR_An</a> | MPU Region Base Address Register Alias <i>n</i> (NS)  |
| 0xE002EDA8 | <a href="#">MPU_RLAR_An</a> | MPU Region Limit Address Register Alias <i>n</i> (NS) |
| 0xE002EDC0 | <a href="#">MPU_MAIR0</a>   | MPU Memory Attribute Indirection Register 0 (NS)      |
| 0xE002EDC4 | <a href="#">MPU_MAIR1</a>   | MPU Memory Attribute Indirection Register 1 (NS)      |

### D1.1.22 Debug Control Block (NS alias)

| Address    | Register              | Description                                       |
|------------|-----------------------|---------------------------------------------------|
| 0xE002EDF0 | <a href="#">DHCSR</a> | Debug Halting Control and Status Register (NS)    |
| 0xE002EDF8 | <a href="#">DCRDR</a> | Debug Core Register Data Register (NS)            |
| 0xE002EDFC | <a href="#">DEMCR</a> | Debug Exception and Monitor Control Register (NS) |

**D1.1.23 Software Interrupt Generation (NS alias)**

| Address    | Register             | Description                                |
|------------|----------------------|--------------------------------------------|
| 0xE002EF00 | <a href="#">STIR</a> | Software Triggered Interrupt Register (NS) |

**D1.1.24 Floating-Point Extension (NS alias)**

| Address    | Register               | Description                                         |
|------------|------------------------|-----------------------------------------------------|
| 0xE002EF34 | <a href="#">FPCCR</a>  | Floating-Point Context Control Register (NS)        |
| 0xE002EF38 | <a href="#">FPCAR</a>  | Floating-Point Context Address Register (NS)        |
| 0xE002EF3C | <a href="#">FPDSCR</a> | Floating-Point Default Status Control Register (NS) |
| 0xE002EF40 | <a href="#">MVFRO</a>  | Media and VFP Feature Register 0 (NS)               |
| 0xE002EF44 | <a href="#">MVFRI</a>  | Media and VFP Feature Register 1 (NS)               |
| 0xE002EF48 | <a href="#">MVFR2</a>  | Media and VFP Feature Register 2 (NS)               |

**D1.1.25 Cache Maintenance Operations (NS alias)**

| Address    | Register                 | Description                                                 |
|------------|--------------------------|-------------------------------------------------------------|
| 0xE002EF50 | <a href="#">ICIALLU</a>  | Instruction Cache Invalidate All to PoU (NS)                |
| 0xE002EF58 | <a href="#">ICIMVAU</a>  | Instruction Cache line Invalidate by Address to PoU (NS)    |
| 0xE002EF5C | <a href="#">DCIMVAC</a>  | Data Cache line Invalidate by Address to PoC (NS)           |
| 0xE002EF60 | <a href="#">DCISW</a>    | Data Cache line Invalidate by Set/Way (NS)                  |
| 0xE002EF64 | <a href="#">DCCMVAU</a>  | Data Cache line Clean by address to PoU (NS)                |
| 0xE002EF68 | <a href="#">DCCMVAC</a>  | Data Cache line Clean by Address to PoC (NS)                |
| 0xE002EF6C | <a href="#">DCCSW</a>    | Data Cache Clean line by Set/Way (NS)                       |
| 0xE002EF70 | <a href="#">DCCIMVAC</a> | Data Cache line Clean and Invalidate by Address to PoC (NS) |
| 0xE002EF74 | <a href="#">DCCISW</a>   | Data Cache line Clean and Invalidate by Set/Way (NS)        |
| 0xE002EF78 | <a href="#">BPIALL</a>   | Branch Predictor Invalidate All (NS)                        |

**D1.1.26 Debug Identification Block (NS alias)**

| Address    | Register                    | Description                                   |
|------------|-----------------------------|-----------------------------------------------|
| 0xE002EFB0 | <a href="#">DLAR</a>        | SCS Software Lock Access Register (NS)        |
| 0xE002EFB4 | <a href="#">DLSR</a>        | SCS Software Lock Status Register (NS)        |
| 0xE002EFB8 | <a href="#">DAUTHSTATUS</a> | Debug Authentication Status Register (NS)     |
| 0xE002EFBC | <a href="#">DDEVARCH</a>    | SCS Device Architecture Register (NS)         |
| 0xE002EFCC | <a href="#">DDEVTYPE</a>    | SCS Device Type Register (NS)                 |
| 0xE002EFD0 | <a href="#">DPIDR4</a>      | SCS Peripheral Identification Register 4 (NS) |
| 0xE002EFD4 | <a href="#">DPIDR5</a>      | SCS Peripheral Identification Register 5 (NS) |
| 0xE002EFD8 | <a href="#">DPIDR6</a>      | SCS Peripheral Identification Register 6 (NS) |
| 0xE002EFD8 | <a href="#">DPIDR7</a>      | SCS Peripheral Identification Register 7 (NS) |
| 0xE002EFE0 | <a href="#">DPIDR0</a>      | SCS Peripheral Identification Register 0 (NS) |
| 0xE002EFE4 | <a href="#">DPIDR1</a>      | SCS Peripheral Identification Register 1 (NS) |
| 0xE002EFE8 | <a href="#">DPIDR2</a>      | SCS Peripheral Identification Register 2 (NS) |
| 0xE002EFEC | <a href="#">DPIDR3</a>      | SCS Peripheral Identification Register 3 (NS) |
| 0xE002EFF0 | <a href="#">DCIDR0</a>      | SCS Component Identification Register 0 (NS)  |

| Address    | Register               | Description                                  |
|------------|------------------------|----------------------------------------------|
| 0xE002EFF4 | <a href="#">DCIDR1</a> | SCS Component Identification Register 1 (NS) |
| 0xE002EFF8 | <a href="#">DCIDR2</a> | SCS Component Identification Register 2 (NS) |
| 0xE002EFFC | <a href="#">DCIDR3</a> | SCS Component Identification Register 3 (NS) |

### D1.1.27 Trace Port Interface Unit

| Address    | Register                     | Description                                    |
|------------|------------------------------|------------------------------------------------|
| 0xE0040000 | <a href="#">TPIU_SSPSR</a>   | TPIU Supported Parallel Port Sizes Register    |
| 0xE0040004 | <a href="#">TPIU_CSPPSR</a>  | TPIU Current Parallel Port Sizes Register      |
| 0xE0040010 | <a href="#">TPIU_ACPR</a>    | TPIU Asynchronous Clock Prescaler Register     |
| 0xE00400F0 | <a href="#">TPIU_SPPR</a>    | TPIU Selected Pin Protocol Register            |
| 0xE0040300 | <a href="#">TPIU_FFSR</a>    | TPIU Formatter and Flush Status Register       |
| 0xE0040304 | <a href="#">TPIU_FFCR</a>    | TPIU Formatter and Flush Control Register      |
| 0xE0040308 | <a href="#">TPIU_PSCR</a>    | TPIU Periodic Synchronization Control Register |
| 0xE0040FB0 | <a href="#">TPIU_LAR</a>     | TPIU Software Lock Access Register             |
| 0xE0040FB4 | <a href="#">TPIU_LSR</a>     | TPIU Software Lock Status Register             |
| 0xE0040FC8 | <a href="#">TPIU_TYPE</a>    | TPIU Device Identifier Register                |
| 0xE0040FCC | <a href="#">TPIU_DEVTYPE</a> | TPIU Device Type Register                      |
| 0xE0040FD0 | <a href="#">TPIU_PIDR4</a>   | TPIU Peripheral Identification Register 4      |
| 0xE0040FD4 | <a href="#">TPIU_PIDR5</a>   | TPIU Peripheral Identification Register 5      |
| 0xE0040FD8 | <a href="#">TPIU_PIDR6</a>   | TPIU Peripheral Identification Register 6      |
| 0xE0040FDC | <a href="#">TPIU_PIDR7</a>   | TPIU Peripheral Identification Register 7      |
| 0xE0040FE0 | <a href="#">TPIU_PIDR0</a>   | TPIU Peripheral Identification Register 0      |
| 0xE0040FE4 | <a href="#">TPIU_PIDR1</a>   | TPIU Peripheral Identification Register 1      |
| 0xE0040FE8 | <a href="#">TPIU_PIDR2</a>   | TPIU Peripheral Identification Register 2      |
| 0xE004FEC  | <a href="#">TPIU_PIDR3</a>   | TPIU Peripheral Identification Register 3      |
| 0xE004FF0  | <a href="#">TPIU_CIDR0</a>   | TPIU Component Identification Register 0       |
| 0xE004FF4  | <a href="#">TPIU_CIDR1</a>   | TPIU Component Identification Register 1       |
| 0xE004FF8  | <a href="#">TPIU_CIDR2</a>   | TPIU Component Identification Register 2       |
| 0xE004FFC  | <a href="#">TPIU_CIDR3</a>   | TPIU Component Identification Register 3       |

## D1.2 Alphabetical list of registers



### D1.2.1 ACTLR, Auxiliary Control Register

The ACTLR characteristics are:

#### Purpose

Provides **IMPLEMENTATION DEFINED** configuration and control options.

#### Usage constraints

Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are **UNPREDICTABLE**.

#### Configurations

This register is always implemented.

#### Attributes

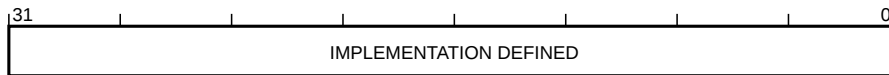
32-bit read/write register located at 0xE000E008.

Secure software can access the Non-secure version of this register via ACTLR\_NS located at 0xE002E008. The location 0xE002E008 is **RES0** to software executing in Non-secure state and the debugger.

It is **IMPLEMENTATION DEFINED** whether this register is banked between Security states.

### Field descriptions

The ACTLR bit assignments are:



#### **IMPLEMENTATION DEFINED, bits [31:0]**

**IMPLEMENTATION DEFINED.** The contents of this field are **IMPLEMENTATION DEFINED**.

## D1.2.2 AFSR, Auxiliary Fault Status Register

The AFSR characteristics are:

### Purpose

Provides **IMPLEMENTATION DEFINED** fault status information.

### Usage constraints

Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are **UNPREDICTABLE**.

### Configurations

This register is always implemented.

### Attributes

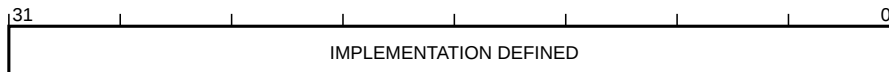
32-bit read/write register located at 0xE000ED3C.

Secure software can access the Non-secure version of this register via AFSR\_NS located at 0xE002ED3C. The location 0xE002ED3C is **RES0** to software executing in Non-secure state and the debugger.

This register is not banked between Security states.

## Field descriptions

The AFSR bit assignments are:



### IMPLEMENTATION DEFINED, bits [31:0]

**IMPLEMENTATION DEFINED**. The contents of this field are **IMPLEMENTATION DEFINED**.

### D1.2.3 AIRCR, Application Interrupt and Reset Control Register

The AIRCR characteristics are:

**Purpose**

Sets or returns interrupt control and reset configuration.

**Usage constraints**

Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are **UNPREDICTABLE**.

**Configurations**

This register is always implemented.

**Attributes**

32-bit read/write register located at 0xE000ED0C.

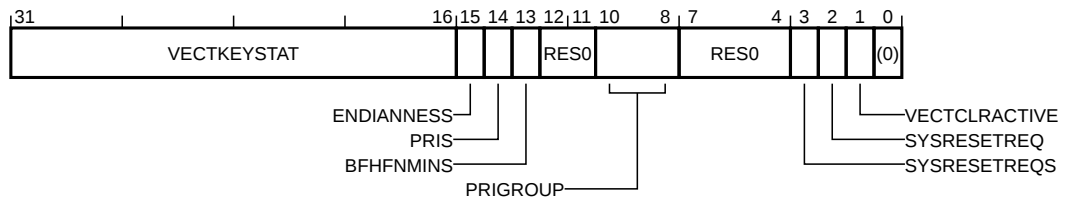
Secure software can access the Non-secure version of this register via AIRCR\_NS located at 0xE002ED0C. The location 0xE002ED0C is **RES0** to software executing in Non-secure state and the debugger.

This register is banked between Security states on a bit by bit basis.

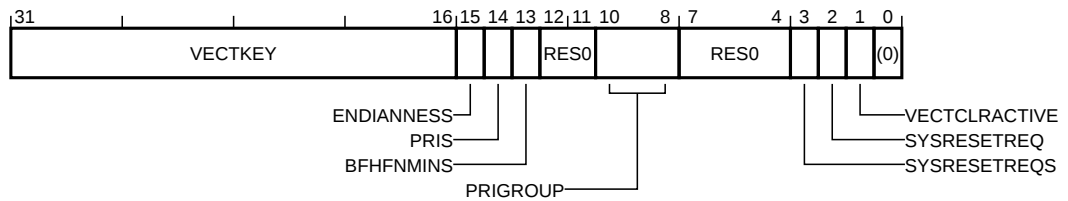
### Field descriptions

The AIRCR bit assignments are:

On a read:



On a write:



**VECTKEY, bits [31:16], on a write**

Vector key. Writes to the AIRCR must be accompanied by a write of the value 0x05FA to this field. Writes to the AIRCR fields that are not accompanied by this value are ignored for the purpose of updating any of the AIRCR values or initiating any AIRCR functionality.

This field is not banked between Security states.

The possible values of this field are:

**0x05FA**

Permit write to AIRCR fields.

**Not 0x05FA**

Accompanying write to AIRCR fields ignored.

**VECTKEYSTAT, bits [31:16], on a read**

Vector key status. Returns the bitwise inverse of the value required to be written to VECTKEY.

This field is not banked between Security states.

This field reads as 0xFA05.

**ENDIANNESS, bit [15]**

Data endianness. Indicates how the PE interprets the memory system data endianness.

This bit is not banked between Security states.

The possible values of this bit are:

**0**

Little-endian.

**1**

Big-endian.

This bit is read-only.

This bit reads as an **IMPLEMENTATION DEFINED** value.

**PRIS, bit [14]**

Prioritize Secure exceptions. The value of this bit defines whether Secure exception priority boosting is enabled.

This bit is not banked between Security states.

The possible values of this bit are:

**0**

Priority ranges of Secure and Non-secure exceptions are identical.

**1**

Non-secure exceptions are de-prioritized.

To allow lock down of this bit, it is **IMPLEMENTATION DEFINED** whether this bit is writable.

This bit is RAZ/WI from Non-secure state.

This bit resets to zero on a Warm reset.

**BFHFNMINs, bit [13]**

BusFault, HardFault, and NMI Non-secure enable. The value of this bit defines whether BusFault and NMI exceptions are Non-secure, and whether exceptions target the Non-secure HardFault exception.

This bit is not banked between Security states.

The possible values of this bit are:

**0**

BusFault, HardFault, and NMI are Secure.

**1**

BusFault and NMI are Non-secure and exceptions can target Non-secure HardFault.

If an implementation resets into Secure state, this bit resets to zero. If an implementation does not support Secure state, this bit is RAO/WI. To allow lock down of this field it is **IMPLEMENTATION DEFINED** whether this bit is writable. The effect of setting both BFHFNMINs and PRIS to 1 is **UNPREDICTABLE**.

This bit is read-only from Non-secure state.

This bit resets to zero on a Warm reset.

**Bits [12:11]**

Reserved, **RES0**.

**PRIGROUP, bits [10:8]**

Priority grouping. The value of this field defines the exception priority binary point position for the selected Security state.

This field is banked between Security states.

The possible values of this field are:

**0b000**

Group priority [7:1], subpriority [0].

**0b001**

Group priority [7:2], subpriority [1:0].

**0b010**

Group priority [7:3], subpriority [2:0].

**0b011**

Group priority [7:4], subpriority [3:0].

**0b100**

Group priority [7:5], subpriority [4:0].

**0b101**

Group priority [7:6], subpriority [5:0].

**0b110**

Group priority [7], subpriority [6:0].

**0b111**

No group priority, subpriority [7:0].

If the Main Extension is not implemented, this field is [RES0](#).

This field resets to zero on a Warm reset.

**Bits [7:4]**

Reserved, [RES0](#).

**SYSRESETREQS, bit [3]**

System reset request Secure only. The value of this bit defines whether the SYSRESETREQ bit is functional for Non-secure use.

This bit is not banked between Security states.

The possible values of this bit are:

**0**

SYSRESETREQ functionality is available to both Security states.

**1**

SYSRESETREQ functionality is only available to Secure state.

This bit is RAZ/WI from Non-secure state.

This bit resets to zero on a Warm reset.

**SYSRESETREQ, bit [2]**

System reset request. This bit allows software or a debugger to request a system reset.

This bit is not banked between Security states.

The possible values of this bit are:

**0**

Do not request a system reset.

**1**

Request a system reset.

When SYSRESETREQS is set to 1, the Non-secure view of this bit is RAZ/WI.

This bit resets to zero on a Warm reset.

**VECTCLRACTIVE, bit [1]**

Clear active state.

A debugger write of one to this bit when the PE is halted in Debug state:

- IPSR is cleared to zero.
- The active state for all Non-secure exceptions is cleared.
- If DHCSR.S\_SDE==1, the active state for all Secure exceptions is cleared.

This bit is not banked between Security states.

The possible values of this bit are:

**0**

Do not clear active state.

**1**

Clear active state.

Writes to this bit while the PE is in Non-debug state are ignored.

This bit reads as zero.

**Bit [0]**

Reserved, [RES0](#).



**0**  
No carry occurred, or last bit shifted was clear.

**1**  
Carry occurred, or last bit shifted was set.

See individual instruction pages for details.

This bit resets to an **UNKNOWN** value on a Warm reset.

**V, bit [28]**

Overflow condition flag. When updated by a flag setting instruction this bit indicates whether a signed overflow occurred.

The possible values of this bit are:

**0**  
Signed overflow did not occur.

**1**  
Signed overflow occurred.

See individual instruction pages for details.

This bit resets to an **UNKNOWN** value on a Warm reset.

**Q, bit [27]**

Sticky saturation flag. When updated by certain instructions this bit indicates either that an overflow occurred or that the result was saturated. This bit is cumulative and can only be cleared to zero by software.

The possible values of this bit are:

**0**  
Saturation or overflow has not occurred since bit was last cleared.

**1**  
Saturation or overflow has occurred since bit was last cleared.

See individual instruction pages for details.

If the Main Extension is not implemented, this bit is **RES0**.

This bit resets to an **UNKNOWN** value on a Warm reset.

**Bits [26:20]**

Reserved, **RES0**.

**GE, bits [19:16]**

Greater than or equal flags. When updated by parallel addition and subtraction instructions these bits record whether the result was greater than or equal to zero. SEL instructions use these bits to determine which register to select a particular byte from.

See individual instruction pages for details.

If the DSP Extension is not implemented, this field is **RES0**.

This field resets to an **UNKNOWN** value on a Warm reset.

**Bits [15:0]**

Reserved, **RES0**.



## D1.2.5 BASEPRI, Base Priority Mask Register

The BASEPRI characteristics are:

### Purpose

Changes the priority level required for exception preemption.

### Usage constraints

Privileged access only. Unprivileged access is RAZ/WI, unless otherwise stated.

### Configurations

Present only if the Main Extension is implemented.

This register is [RES0](#) if the Main Extension is not implemented.

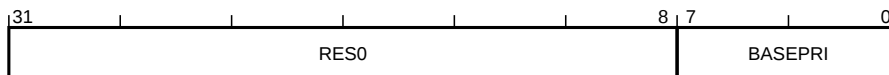
### Attributes

32-bit read/write special-purpose register.

This register is banked between Security states.

## Field descriptions

The BASEPRI bit assignments are:



### Bits [31:8]

Reserved, [RES0](#).

### BASEPRI, bits [7:0]

Base priority mask. BASEPRI changes the priority level required for exception preemption. It has an effect only when BASEPRI has a lower value than the unmasked priority level of the currently executing software.

The possible values of this field are:

**0**

Disables masking by BASEPRI.

**1-255**

Priority value.

The number of implemented bits in BASEPRI is the same as the number of implemented bits in each field of the priority registers, and BASEPRI has the same format as those fields.

This field resets to zero on a Warm reset.

## D1.2.6 BFAR, BusFault Address Register

The BFAR characteristics are:

### Purpose

Shows the address associated with a precise data access BusFault.

### Usage constraints

Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are **UNPREDICTABLE**.

### Configurations

Present only if the Main Extension is implemented.

This register is **RES0** if the Main Extension is not implemented.

### Attributes

32-bit read/write register located at 0xE000ED38.

Secure software can access the Non-secure version of this register via BFAR\_NS located at 0xE002ED38. The location 0xE002ED38 is **RES0** to software executing in Non-secure state and the debugger.

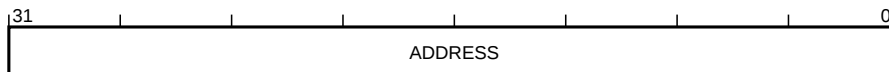
This register is not banked between Security states.

### Preface

The Non-secure version of this register is RAZ/WI if AIRCR.BFHFNMINs is set to 0.

### Field descriptions

The BFAR bit assignments are:



### ADDRESS, bits [31:0]

Data address for a precise BusFault. This register is updated with the address of a location that produced a BusFault. BFSR shows the reason for the fault. This field is valid only when BFSR.BFARVALID is set, otherwise it is **UNKNOWN**.

In implementations without unique BFAR and MMFAR registers, the value of this register is **UNKNOWN** if MMFSR.MMARVALID is set.

If AIRCR.BFHFNMINs is zero this field is RAZ/WI from Non-secure state.

This field resets to an **UNKNOWN** value on a Warm reset.

### Note

If an implementation shares a common BFAR and MMFAR it must not leak Secure state information to the Non-secure state. One possible implementation is that BFAR shares resource with the Secure MMFAR if AIRCR.BFHFNMINs is zero, and with the Non-secure MMFAR if AIRCR.BFHFNMINs is set.

## D1.2.7 BFSR, BusFault Status Register

The BFSR characteristics are:

### Purpose

Shows the status of bus errors resulting from instruction fetches and data accesses.

### Usage constraints

Privileged access permitted only. Unprivileged accesses generate a fault.

If the Main Extension is implemented, this register is word, halfword, and byte accessible.

If the Main Extension is not implemented, this register is word accessible only, halfword and byte accesses are **UNPREDICTABLE**.

### Configurations

Present only if the Main Extension is implemented.

This register is **RES0** if the Main Extension is not implemented.

### Attributes

8-bit read/write-one-to-clear register located at 0xE000ED29.

Secure software can access the Non-secure version of this register via BFSR\_NS located at 0xE002ED29. The location 0xE002ED29 is **RES0** to software executing in Non-secure state and the debugger.

This register is not banked between Security states.

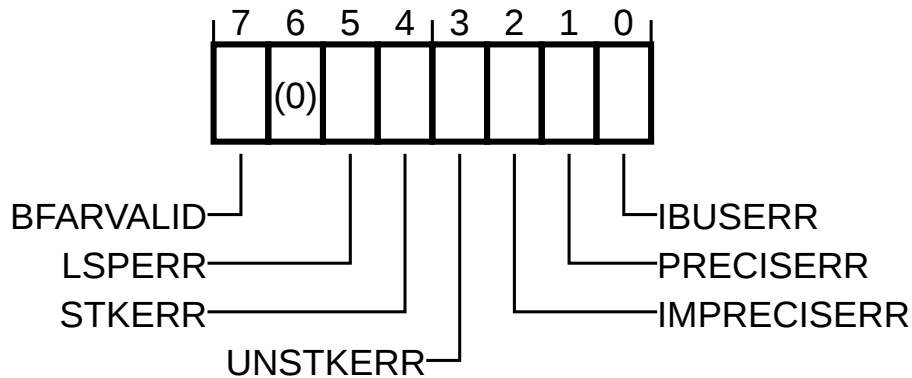
This register is part of CFSR.

### Preface

The Non-secure version of this register is RAZ/WI if AIRCR.BFHFNMINS is set to 0.

### Field descriptions

The BFSR bit assignments are:



### BFARVALID, bit [7]

BFAR valid. Indicates validity of the contents of the BFAR register.

The possible values of this bit are:

**0**

BFAR content not valid.

**1**

BFAR content valid.

If AIRCR.BFHFNMINs is zero this bit is RAZ/WI from Non-secure state.

This bit resets to zero on a Warm reset.

**Bit [6]**

Reserved, RES0.

**LSPERR, bit [5]**

Lazy state preservation error. Records whether a precise BusFault occurred during FP lazy state preservation.

The possible values of this bit are:

**0**

No BusFault occurred.

**1**

BusFault occurred.

If AIRCR.BFHFNMINs is zero this bit is RAZ/WI from Non-secure state.

This bit resets to zero on a Warm reset.

**STKERR, bit [4]**

Stack error. Records whether a precise derived BusFault occurred during exception entry stacking.

The possible values of this bit are:

**0**

No derived BusFault occurred.

**1**

Derived BusFault occurred during exception entry.

If AIRCR.BFHFNMINs is zero this bit is RAZ/WI from Non-secure state.

This bit resets to zero on a Warm reset.

**UNSTKERR, bit [3]**

Unstack error. Records whether a precise derived BusFault occurred during exception return unstacking.

The possible values of this bit are:

**0**

No derived BusFault occurred.

**1**

Derived BusFault occurred during exception return.

If AIRCR.BFHFNMINs is zero this bit is RAZ/WI from Non-secure state.

This bit resets to zero on a Warm reset.

**IMPRECISERR, bit [2]**

Imprecise error. Records whether an imprecise data access error has occurred.

The possible values of this bit are:

**0**

No imprecise data access error has occurred.

**1**

Imprecise data access error has occurred.

If AIRCR.BFHFNMINs is zero this bit is RAZ/WI from Non-secure state.

This bit resets to zero on a Warm reset.

**PRECISERR, bit [1]**

Precise error. Records whether a precise data access error has occurred.

The possible values of this bit are:

**0**

No precise data access error has occurred.

**1**

Precise data access error has occurred.

When a precise error is recorded, the associated address is written to the BFAR and BFSR.BFARVALID bit is set.

If AIRCR.BFHFNMINs is zero this bit is RAZ/WI from Non-secure state.

This bit resets to zero on a Warm reset.

**IBUSERR, bit [0]**

Instruction bus error. Records whether a precise BusFault on an instruction prefetch has occurred.

The possible values of this bit are:

**0**

No BusFault on instruction prefetch has occurred.

**1**

A BusFault on an instruction prefetch has occurred.

An IBUSERR is only recorded if the instruction is issued for execution.

If AIRCR.BFHFNMINs is zero this bit is RAZ/WI from Non-secure state.

This bit resets to zero on a Warm reset.

### D1.2.8 BPIALL, Branch Predictor Invalidate All

The BPIALL characteristics are:

**Purpose**

Invalidate all entries from branch predictors.

**Usage constraints**

Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are **UNPREDICTABLE**.

**Configurations**

This register is always implemented.

**Attributes**

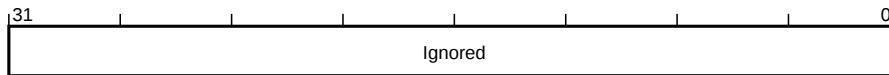
32-bit write-only register located at 0xE000EF78.

Secure software can access the Non-secure version of this register via BPIALL\_NS located at 0xE002EF78. The location 0xE002EF78 is **RES0** to software executing in Non-secure state and the debugger.

This register is not banked between Security states.

### Field descriptions

The BPIALL bit assignments are:



**Ignored, bits [31:0]**

Ignored. The value written to this field is ignored.

## D1.2.9 CCR, Configuration and Control Register

The CCR characteristics are:

### Purpose

Sets or returns configuration and control data.

### Usage constraints

Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are **UNPREDICTABLE**.

### Configurations

This register is always implemented.

### Attributes

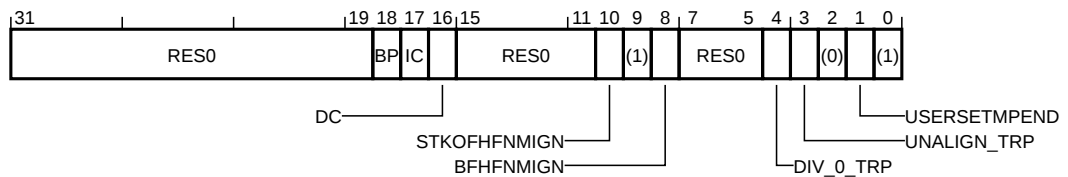
32-bit read/write register located at 0xE000ED14.

Secure software can access the Non-secure version of this register via CCR\_NS located at 0xE002ED14. The location 0xE002ED14 is **RES0** to software executing in Non-secure state and the debugger.

This register is banked between Security states on a bit by bit basis.

## Field descriptions

The CCR bit assignments are:



### Bits [31:19]

Reserved, **RES0**.

### BP, bit [18]

Branch prediction enable. Enables program flow prediction for the selected Security state.

This bit is banked between Security states.

The possible values of this bit are:

**0**

Program flow prediction disabled for the selected Security state.

**1**

Program flow prediction enabled for the selected Security state.

If program flow prediction cannot be disabled, this bit is RAO/WI. If the program flow prediction is not supported, this bit is RAZ/WI.

If the Main Extension is not implemented, this bit is **RES0**.

This bit resets to zero on a Warm reset.

### IC, bit [17]

Instruction cache enable. This is a global enable bit for instruction caches in the selected Security state.

This bit is banked between Security states.

The possible values of this bit are:

**0**  
Instruction caches disabled for the selected Security state.

**1**  
Instruction caches enabled for the selected Security state.  
If the PE does not implement instruction caches, this bit is RAZ/WI.  
If the Main Extension is not implemented, this bit is [RES0](#).  
This bit resets to zero on a Warm reset.

**DC, bit [16]**

Data cache enable. Enables data caching of all data accesses to Normal memory.  
This bit is banked between Security states.  
The possible values of this bit are:

**0**  
Data caching disabled.

**1**  
Data caching enabled.

The secure version of this bit controls the Cacheability of accesses to secure memory.  
The non-secure version of this bit controls the Cacheability of accesses to non-secure memory.  
If the Main Extension is not implemented, this bit is [RES0](#).  
This bit resets to zero on a Warm reset.

**Bits [15:11]**

Reserved, [RES0](#).

**STKOFHFNMIGN, bit [10]**

Stack overflow in HardFault and NMI ignore. Controls the effect of a stack limit violation while executing at a requested priority less than 0 for the Security state with which the stack limit register is associated.

This bit is banked between Security states.

The possible values of this bit are:

**0**  
Stack limit faults not ignored.

**1**  
Stack limit faults at requested priorities of less than 0 ignored.

If the Main Extension is not implemented, this bit is [RES0](#).

This bit resets to zero on a Warm reset.

**Bit [9]**

Reserved, [RES1](#).

**BFHFNMIGN, bit [8]**

BusFault in HardFault or NMI ignore. Determines the effect of precise BusFaults on handlers running at a requested priority less than 0.

This bit is not banked between Security states.

The possible values of this bit are:

**0**  
Precise BusFaults not ignored.



**1**

Precise BusFaults at requested priorities of less than 0 ignored.

If AIRCR.BFHFNMIN is 0, this bit is read-only from Non-secure state.

If the Main Extension is not implemented, this bit is **RES0**.

This bit resets to zero on a Warm reset.

**Bits [7:5]**

Reserved, **RES0**.

**DIV\_0\_TRP, bit [4]**

Divide by zero trap. Controls the generation of a DIVBYZERO UsageFault when attempting to perform integer division by zero.

This bit is banked between Security states.

The possible values of this bit are:

**0**

DIVBYZERO UsageFault generation disabled.

**1**

DIVBYZERO UsageFault generation enabled.

If the Main Extension is not implemented, this bit is **RES0**.

This bit resets to zero on a Warm reset.

**UNALIGN\_TRP, bit [3]**

Unaligned trap. Controls the trapping of unaligned word or halfword accesses.

This bit is banked between Security states.

The possible values of this bit are:

**0**

Unaligned accesses permitted from LDR, LDRH, STR, and STRH.

**1**

Any unaligned transaction generates an UNALIGNED UsageFault.

Unaligned load/store multiples and atomic/exclusive accesses always generate an UNALIGNED UsageFault.

If the Main Extension is not implemented, this bit is **RES1**.

This bit resets to zero on a Warm reset if the Main Extension is implemented.

**Bit [2]**

Reserved, **RES0**.

**USERSETMPEND, bit [1]**

User set main pending. Determines whether unprivileged accesses are permitted to pend interrupts via the STIR.

This bit is banked between Security states.

The possible values of this bit are:

**0**

Unprivileged accesses to the STIR generate a fault.

**1**

Unprivileged accesses to the STIR are permitted.

If the Main Extension is not implemented, this bit is [RES0](#).

This bit resets to zero on a Warm reset.

**Bit [0]**

Reserved, [RES1](#).

### D1.2.10 CCSIDR, Current Cache Size ID register

The CCSIDR characteristics are:

#### Purpose

The CCSIDR provides information about the architecture of the currently selected cache.

#### Usage constraints

Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are **UNPREDICTABLE**.

If CSSELR points to an unimplemented cache, the value of this register is **UNKNOWN**.

#### Configurations

This register is always implemented.

#### Attributes

32-bit read-only register located at 0xE000ED80.

Secure software can access the Non-secure version of this register via CCSIDR\_NS located at 0xE002ED80. The location 0xE002ED80 is **RES0** to software executing in Non-secure state and the debugger.

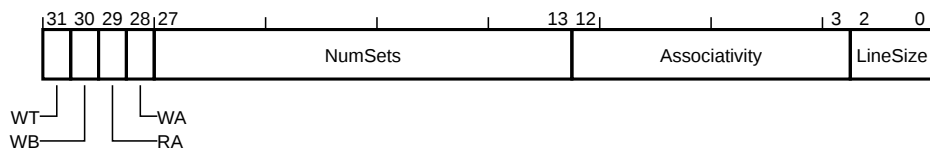
This register is banked between Security states.

### Preface

Provides indirect read access to the architecture of the cache currently selected by CSSELR. The parameters NumSets, Associativity, and LineSize in these registers define the architecturally visible parameters that are required for the cache maintenance by Set/Way instructions. They are not guaranteed to represent the actual microarchitectural features of a design. You cannot make any inference about the actual sizes of caches based on these parameters.

### Field descriptions

The CCSIDR bit assignments are:



#### WT, bit [31]

Write-Through. Indicates whether the currently selected cache level supports Write-Through.

The possible values of this bit are:

**0**  
Not supported.

**1**  
Supported.

This bit reads as an **IMPLEMENTATION DEFINED** value.

#### WB, bit [30]

Writeback. Indicates whether the currently selected cache level supports Write-Back.

The possible values of this bit are:

**0**  
Not supported.

**1**  
Supported.

This bit reads as an **IMPLEMENTATION DEFINED** value.

**RA, bit [29]**

Read-allocate. Indicates whether the currently selected cache level supports read-allocation.

The possible values of this bit are:

**0**  
Not supported.

**1**  
Supported.

This bit reads as an **IMPLEMENTATION DEFINED** value.

**WA, bit [28]**

Write-Allocate. Indicates whether the currently selected cache level supports write-allocation.

The possible values of this bit are:

**0**  
Not supported.

**1**  
Supported.

This bit reads as an **IMPLEMENTATION DEFINED** value.

**NumSets, bits [27:13]**

Number of sets. Indicates (Number of sets in the currently selected cache) - 1. Therefore, a value of 0 indicates that 1 is set in the cache. The number of sets does not have to be a power of 2.

This field reads as an **IMPLEMENTATION DEFINED** value.

**Associativity, bits [12:3]**

Associativity. Indicates (Associativity of cache) - 1. A value of 0 indicates an associativity of 1. The associativity does not have to be a power of 2.

This field reads as an **IMPLEMENTATION DEFINED** value.

**LineSize, bits [2:0]**

Line size. Indicates ( $\log_2(\text{Number of words per line in the currently selected cache}) - 2$ ).

This field reads as an **IMPLEMENTATION DEFINED** value.

## D1.2.11 CFSR, Configurable Fault Status Register

The CFSR characteristics are:

### Purpose

Contains the three Configurable Fault Status Registers.

### Usage constraints

Privileged access permitted only. Unprivileged accesses generate a fault.

If the Main Extension is implemented, this register is word, halfword, and byte accessible.

If the Main Extension is not implemented, this register is word accessible only, halfword and byte accesses are **UNPREDICTABLE**.

### Configurations

Present only if the Main Extension is implemented.

This register is **RES0** if the Main Extension is not implemented.

### Attributes

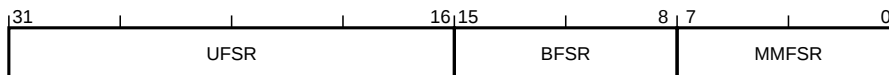
32-bit read/write-one-to-clear register located at 0xE000ED28.

Secure software can access the Non-secure version of this register via CFSR\_NS located at 0xE002ED28. The location 0xE002ED28 is **RES0** to software executing in Non-secure state and the debugger.

This register is banked between Security states on a bit by bit basis.

## Field descriptions

The CFSR bit assignments are:



### UFSR, bits [31:16]

UsageFault Status Register. Provides information on UsageFault exceptions.

This field is banked between Security states.

See UFSR.

This field resets to zero on a Warm reset.

### BFSR, bits [15:8]

BusFault Status Register. Provides information on BusFault exceptions.

This field is not banked between Security states.

See BFSR.

This field resets to zero on a Warm reset.

### MMFSR, bits [7:0]

MemManage Fault Status Register. Provides information on MemManage exceptions.

This field is banked between Security states.

See MMFSR.

This field resets to zero on a Warm reset.

## D1.2.12 CLIDR, Cache Level ID Register

The CLIDR characteristics are:

### Purpose

Identifies the type of caches implemented and the level of coherency and unification.

### Usage constraints

Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are **UNPREDICTABLE**.

### Configurations

This register is always implemented.

### Attributes

32-bit read-only register located at 0xE000ED78.

Secure software can access the Non-secure version of this register via CLIDR\_NS located at 0xE002ED78. The location 0xE002ED78 is **RES0** to software executing in Non-secure state and the debugger.

This register is not banked between Security states.

## Field descriptions

The CLIDR bit assignments are:

|     |      |     |       |        |        |        |        |        |        |        |    |    |    |    |   |   |   |   |   |   |   |
|-----|------|-----|-------|--------|--------|--------|--------|--------|--------|--------|----|----|----|----|---|---|---|---|---|---|---|
| 31  | 30   | 29  | 27    | 26     | 24     | 23     | 21     | 20     | 18     | 17     | 15 | 14 | 12 | 11 | 9 | 8 | 6 | 5 | 3 | 2 | 0 |
| ICB | LoUU | LoC | LoUIS | Ctype7 | Ctype6 | Ctype5 | Ctype4 | Ctype3 | Ctype2 | Ctype1 |    |    |    |    |   |   |   |   |   |   |   |

### ICB, bits [31:30]

Inner cache boundary. This field indicates the boundary between inner and outer domain.

The possible values of this field are:

#### 0b00

Not disclosed in this mechanism.

#### 0b01

L1 cache is the highest inner level.

#### 0b10

L2 cache is the highest inner level.

#### 0b11

L3 cache is the highest inner level.

This field reads as an **IMPLEMENTATION DEFINED** value.

### LoUU, bits [29:27]

Level of Unification Uniprocessor. This field indicates the Level of Unification Uniprocessor for the cache hierarchy.

This field reads as an **IMPLEMENTATION DEFINED** value.

### LoC, bits [26:24]

Level of Coherence. This field indicates the Level of Coherence for the cache hierarchy.

This field reads as an **IMPLEMENTATION DEFINED** value.

**LoUIS, bits [23:21]**

Level of Unification Inner Shareable. This field indicates the Level of Unification Shareable for the cache hierarchy.

This field reads as an **IMPLEMENTATION DEFINED** value.

**Ctypem, bits [3(m-1)+2:3(m-1)], for m = 1 to 7**

Cache type field *m*. Indicates the type of cache implemented at level *m*.

The possible values of this field are:

**0b000**

No cache.

**0b001**

Instruction cache only.

**0b010**

Data cache only.

**0b011**

Separate instruction and data caches.

**0b100**

Unified cache.

All other values are reserved.

If Ctype<*m*> is set to 0b000, and *m* < 7, then all of the following apply.

Level *m* represents the last level of software-visible cache.

Ctype<*m*=1> through to Ctype7 must read as zero.

Software must treat Ctype<*m*+1> through Ctype7 as if they are invalid and read as an **UNKNOWN** value.

This field reads as an **IMPLEMENTATION DEFINED** value.

### D1.2.13 CONTROL, Control Register

The CONTROL characteristics are:

**Purpose**

Provides access to the PE control fields.

**Usage constraints**

Privileged access only, but unprivileged writes are ignored unless otherwise specified.

**Configurations**

This register is always implemented.

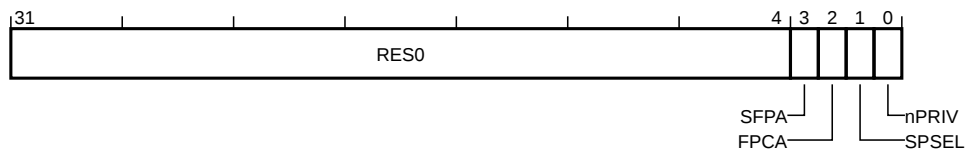
**Attributes**

32-bit read/write special-purpose register.

This register is banked between Security states on a bit by bit basis.

### Field descriptions

The CONTROL bit assignments are:



**Bits [31:4]**

Reserved, [RES0](#).

**SFPA, bit [3]**

Secure floating-point active. Indicates that the floating-point registers contain active state that belongs to the Secure state.

This bit is not banked between Security states.

The possible values of this bit are:

**0**

The floating-point registers do not contain state that belongs to the Secure state.

**1**

The floating-point registers contain state that belongs to the Secure state.

This bit is RAZ/WI from Non-secure state.

If the Security Extension is not implemented, this bit is [RES0](#).

If the Floating-point Extension is not implemented, this bit is [RES0](#).

This bit resets to zero on a Warm reset.

**FPCA, bit [2]**

Floating-point context active. Defines whether the FP Extension is active in the current context.

This bit is not banked between Security states.

The possible values of this bit are:

**0**

FP Extension is not active.



**1**

FP Extension is active.

When NSACR.CP10 is set to zero, the Non-secure view of this bit is read-only. If FPCCR.ASPEN is set to 1, enabling automatic floating-point state preservation, then the PE sets this bit to 1 on successful completion of any floating-point instruction.

If the Floating-point Extension is not implemented, this bit is [RES0](#).

This bit resets to zero on a Warm reset.

**SPSEL, bit [1]**

Stack-pointer select. Defines the stack pointer to be used.

This bit is banked between Security states.

The possible values of this bit are:

**0**

Use SP\_main as the current stack.

**1**

In Thread mode use SP\_process as the current stack.

This bit resets to zero on a Warm reset.

**nPRIV, bit [0]**

Not privileged. Defines the execution privilege in Thread mode.

This bit is banked between Security states.

The possible values of this bit are:

**0**

Thread mode has privileged access.

**1**

Thread mode has unprivileged access only.

If the Main Extension is not implemented, it is [IMPLEMENTATION DEFINED](#) whether this field is RW or RAZ/WI.

This bit resets to zero on a Warm reset.

### D1.2.14 CPACR, Coprocessor Access Control Register

The CPACR characteristics are:

**Purpose**

Specifies the access privileges for coprocessors and the Floating-point Extension.

**Usage constraints**

Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are **UNPREDICTABLE**.

**Configurations**

Present only if the Main Extension is implemented.

This register is **RES0** if the Main Extension is not implemented.

**Attributes**

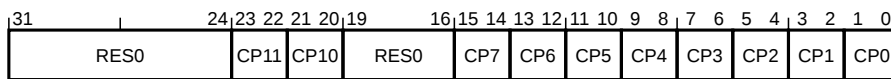
32-bit read/write register located at 0xE000ED88.

Secure software can access the Non-secure version of this register via CPACR\_NS located at 0xE002ED88. The location 0xE002ED88 is **RES0** to software executing in Non-secure state and the debugger.

This register is banked between Security states.

### Field descriptions

The CPACR bit assignments are:



**Bits [31:24]**

Reserved, **RES0**.

**CP11, bits [23:22]**

CP11 Privilege. The value in this field is ignored. If the implementation does not include the FP Extension, this field is RAZ/WI. If the value of this bit is not programmed to the same value as the CP10 field, then the value is **UNKNOWN**.

This field resets to an **UNKNOWN** value on a Warm reset.

**CP10, bits [21:20]**

CP10 Privilege. Defines the access rights for the floating-point functionality.

The possible values of this field are:

**0b00**

All accesses to the FP Extension result in NOCP UsageFault.

**0b01**

Unprivileged accesses to the FP Extension result in NOCP UsageFault.

**0b11**

Full access to the FP Extension.

All other values are reserved.

The features controlled by this field are:

The execution of any instructions within the encoding space defined by IsCPInstruction().

Access to any floating-point registers in the range D0-D16.

If the implementation does not include the Floating-point Extension, this field is RAZ/WI. See individual floating-point instruction pages for details.

This field resets to an **UNKNOWN** value on a Warm reset.

**Bits [19:16]**

Reserved, **RES0**.

**CPm, bits [2m+1:2m], for m = 0 to 7**

Coprocessor *m* privilege. Controls access privileges for coprocessor *m*.

The possible values of this field are:

**0b00**

Access denied. Any attempted access generates a NOCP UsageFault.

**0b01**

Privileged access only. An unprivileged access generates a NOCP UsageFault.

**0b10**

Reserved.

**0b11**

Full access.

If coprocessor *m* is not implemented, this field is RAZ/WI.

This field resets to an **UNKNOWN** value on a Warm reset.

### D1.2.15 CPPWR, Coprocessor Power Control Register

The CPPWR characteristics are:

**Purpose**

Specifies whether coprocessors are permitted to enter a non-retentive power state.

**Usage constraints**

Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are **UNPREDICTABLE**.

**Configurations**

Present only if the Main Extension is implemented.

This register is **RES0** if the Main Extension is not implemented.

**Attributes**

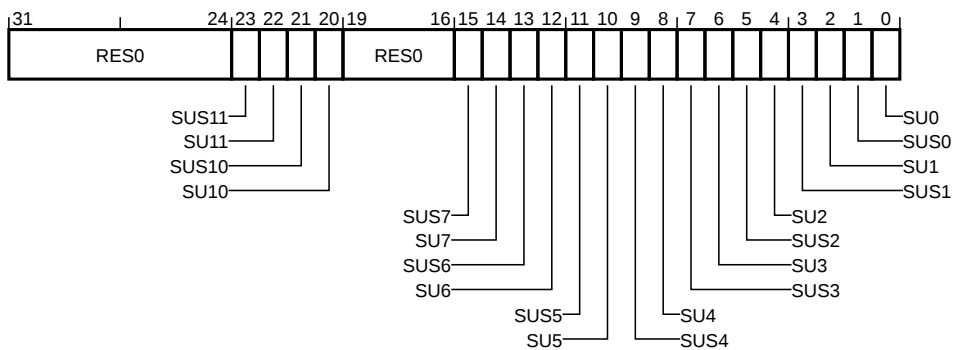
32-bit read/write register located at 0xE000E00C.

Secure software can access the Non-secure version of this register via CPPWR\_NS located at 0xE002E00C. The location 0xE002E00C is **RES0** to software executing in Non-secure state and the debugger.

This register is not banked between Security states.

### Field descriptions

The CPPWR bit assignments are:



**Bits [31:24]**

Reserved, **RES0**.

**SUS11, bit [23]**

State **UNKNOWN** Secure only 11. The value in this field is ignored. If the value of this bit is not programmed to the same value as the SUS10 field, then the value is **UNKNOWN**.

If SU10 is always RAZ/WI this field is also RAZ/WI.

This bit is RAZ/WI from Non-secure state.

This bit resets to zero on a Warm reset.

**SU11, bit [22]**

State **UNKNOWN** 11. The value in this field is ignored. If the value of this bit is not programmed to the same value as the SU10 field, then the value is **UNKNOWN**.

When SUS10 is set to 1, the Non-secure view of this bit is RAZ/WI. If SU10 is always RAZ/WI this field is also RAZ/WI.

This bit resets to zero on a Warm reset.

**SUS10, bit [21]**

State **UNKNOWN** Secure only 10. This bit indicates and allows modification of whether the SU10 field can be modified from Non-secure state.

The possible values of this bit are:

**0**

The SU10 field is accessible from both Security states.

**1**

The SU10 field is only accessible from the Secure state.

If SU10 is always RAZ/WI this field is also RAZ/WI.

This bit is RAZ/WI from Non-secure state.

This bit resets to zero on a Warm reset.

**SU10, bit [20]**

State **UNKNOWN** 10. This bit indicates and allows modification of whether the state associated with the floating-point unit is permitted to become **UNKNOWN**. This can be used as a hint to power control logic that the floating-point unit might be powered down.

The possible values of this bit are:

**0**

The floating-point state is not permitted to become **UNKNOWN**.

**1**

The floating-point state is permitted to become **UNKNOWN**.

When SUS10 is set to 1, the Non-secure view of this bit is RAZ/WI. It is **IMPLEMENTATION DEFINED** whether this bit is always RAZ/WI.

This bit resets to zero on a Warm reset.

**Bits [19:16]**

Reserved, **RES0**.

**SUS $m$ , bit [2 $m$ +1], for  $m = 0$  to 7**

State **UNKNOWN** Secure only  $m$ . This field indicates and allows modification of whether the SU $m$  field can be modified from Non-secure state.

The possible values of this field are:

**0**

The SU $m$  field is accessible from both Security states.

**1**

The SU $m$  field is only accessible from the Secure state.

If SU $m$  is always RAZ/WI this field is also RAZ/WI.

This field is RAZ/WI from Non-secure state.

This field resets to zero on a Warm reset.

**SU $m$ , bit [2 $m$ ], for  $m = 0$  to 7**

State **UNKNOWN**  $m$ . This field indicates and allows modification of whether the state associated with coprocessor  $m$  is permitted to become **UNKNOWN**. This can be used as a hint to power control logic that the coprocessor might be powered down.

The possible values of this field are:

**0**

The coprocessor state is not permitted to become **UNKNOWN**.

**1**

The coprocessor state is permitted to become **UNKNOWN**.

When  $SUSm$  is set to 1, the Non-secure view of this bit is RAZ/WI. It is **IMPLEMENTATION DEFINED** whether this bit is always RAZ/WI.

This field resets to zero on a Warm reset.

## D1.2.16 CPUID, CPUID Base Register

The CPUID characteristics are:

### Purpose

Provides identification information for the PE, including an implementer code for the device and a device ID number.

### Usage constraints

Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are [UNPREDICTABLE](#).

### Configurations

This register is always implemented.

### Attributes

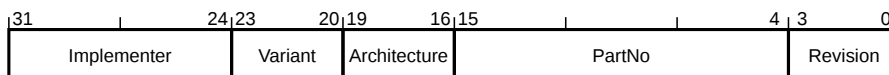
32-bit read-only register located at 0xE000ED00.

Secure software can access the Non-secure version of this register via CPUID\_NS located at 0xE002ED00. The location 0xE002ED00 is [RES0](#) to software executing in Non-secure state and the debugger.

This register is not banked between Security states.

## Field descriptions

The CPUID bit assignments are:



### Implementer, bits [31:24]

Implementer code. This field must hold an implementer code that has been assigned by Arm.

The possible values of this field are:

#### 0x41

'A': Arm Limited.

#### Not 0x41

Implementer other than Arm Limited.

Arm can assign codes that are not published in this manual. All values not assigned by Arm are reserved and must not be used.

This field reads as an [IMPLEMENTATION DEFINED](#) value.

### Variant, bits [23:20]

Variant number. [IMPLEMENTATION DEFINED](#) variant number. Typically, this field is used to distinguish between different product variants, or major revisions of a product.

This field reads as an [IMPLEMENTATION DEFINED](#) value.

### Architecture, bits [19:16]

Architecture version. Defines the Architecture implemented by the PE.

The possible values of this field are:

#### 0b1100

Armv8-M architecture without Main Extension.

#### 0b1111

Armv8-M architecture with Main Extension.

All other values are reserved.

This field reads as an **IMPLEMENTATION DEFINED** value.

**PartNo, bits [15:4]**

Part number. **IMPLEMENTATION DEFINED** primary part number for the device.

This field reads as an **IMPLEMENTATION DEFINED** value.

**Revision, bits [3:0]**

Revision number. **IMPLEMENTATION DEFINED** revision number for the device.

This field reads as an **IMPLEMENTATION DEFINED** value.



## D1.2.17 CSSELR, Cache Size Selection Register

The CSSELR characteristics are:

### Purpose

Selects the current Cache Size ID Register, CCSIDR, by specifying the required cache level and the cache type (either instruction or data cache)

### Usage constraints

Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are **UNPREDICTABLE**.

### Configurations

This register is always implemented.

### Attributes

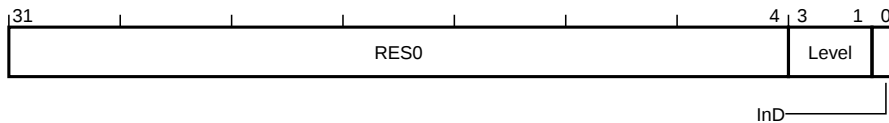
32-bit read/write register located at 0xE000ED84.

Secure software can access the Non-secure version of this register via CSSELR\_NS located at 0xE002ED84. The location 0xE002ED84 is **RES0** to software executing in Non-secure state and the debugger.

This register is banked between Security states.

## Field descriptions

The CSSELR bit assignments are:



### Bits [31:4]

Reserved, **RES0**.

### Level, bits [3:1]

Cache level. Selects which cache level is to be identified. Permitted values are from 0b000, indicating Level 1 cache, to 0b110 indicating Level 7 cache.

The possible values of this field are:

#### 0b000

Level 1 cache.

#### 0b001

Level 2 cache.

#### 0b010

Level 3 cache.

#### 0b011

Level 4 cache.

#### 0b100

Level 5 cache.

#### 0b101

Level 6 cache.

#### 0b110

Level 7 cache.

All other values are reserved.

Writing a reserved value or value corresponding to an unimplemented level of cache is **CONSTRAINED UNPREDICTABLE**.

This field resets to an **UNKNOWN** value on a Warm reset.

**InD, bit [0]**

Instruction not data. Selects whether the instruction or the data cache is to be identified.

The possible values of this bit are:

**0**

Data or unified cache.

**1**

Instruction cache.

This bit resets to an **UNKNOWN** value on a Warm reset.

## D1.2.18 CTR, Cache Type Register

The CTR characteristics are:

### Purpose

Provides information about the architecture of the caches.

### Usage constraints

Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are [UNPREDICTABLE](#).

### Configurations

This register is always implemented.

### Attributes

32-bit read-only register located at 0xE000ED7C.

Secure software can access the Non-secure version of this register via CTR\_NS located at 0xE002ED7C. The location 0xE002ED7C is [RES0](#) to software executing in Non-secure state and the debugger.

This register is banked between Security states.

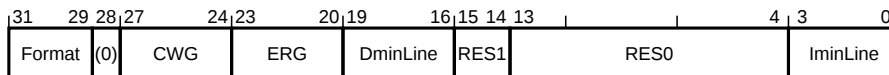
## Field descriptions

The CTR bit assignments are:

### When Format!='0b100':



### When Format=='0b100':



### Format, bits [31:29]

Cache Type Register format. Indicates whether cache type information is provided.

The possible values of this field are:

#### 0b000

No cache type information is provided.

#### 0b100

Cache type information is provided.

All other values are reserved.

The value of this field is an [IMPLEMENTATION DEFINED](#) choice of either 0b000 or 0b100.

If CLIDR is nonzero then this field must read as 0b100.

This field reads as an [IMPLEMENTATION DEFINED](#) value.

### Bits [28:0], when Format!='0b100'

Reserved, [RES0](#).

### Bit [28], when Format=='0b100'

Reserved, [RES0](#).

**CWG, bits [27:24], when Format=='0b100'**

Cache Write-Back Granule.  $\log_2$  of the number of words of the maximum size of memory that can be overwritten as a result of the eviction of a cache entry that has had a memory location in it modified.

The possible values of this field are:

**0b0000**

Indicates that this register does not provide Cache Write-Back Granule information and either the architectural maximum of 512 words (2KB) must be assumed, or the Cache Write-Back Granule can be determined from maximum cache line size encoded in the Cache Size ID Registers.

**0b0001-0b1000**

$\log_2$  of the number of words.

All other values are reserved.

This field reads as an **IMPLEMENTATION DEFINED** value.

**ERG, bits [23:20], when Format=='0b100'**

Exclusives Reservation Granule.  $\log_2$  of the number of words of the maximum size of the reservation granule that has been implemented for the Load-Exclusive and Store-Exclusive instructions.

The possible values of this field are:

**0b0000**

Indicates that this register does not provide Exclusives Reservation Granule information and the architectural maximum of 512 words (2KB) must be assumed.

**0b0001-0b1000**

$\log_2$  of the number of words.

All other values are reserved.

This field reads as an **IMPLEMENTATION DEFINED** value.

**DminLine, bits [19:16], when Format=='0b100'**

Data cache minimum line length.  $\log_2$  of the number of words in the smallest cache line of all the data caches and unified caches that are controlled by the PE.

This field reads as an **IMPLEMENTATION DEFINED** value.

**Bits [15:14], when Format=='0b100'**

Reserved, **RES1**.

**Bits [13:4], when Format=='0b100'**

Reserved, **RES0**.

**IminLine, bits [3:0], when Format=='0b100'**

Instruction cache minimum line length.  $\log_2$  of the number of words in the smallest cache line of all the instruction caches that are controlled by the PE.

This field reads as an **IMPLEMENTATION DEFINED** value.

## D1.2.19 DAUTHCTRL, Debug Authentication Control Register

The DAUTHCTRL characteristics are:

### Purpose

This register allows the **IMPLEMENTATION DEFINED** authentication interface to be overridden from software.

### Usage constraints

Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are **UNPREDICTABLE**.

This register is **RES0** if accessed via the debugger.

### Configurations

Present if Halting debug or the Main Extension is implemented.

This register is **RES0** if both Halting debug and Main Extension are not implemented.

Present only if the Security Extension is implemented.

This register is **RES0** if the Security Extension is not implemented.

### Attributes

32-bit read/write register located at 0xE000EE04.

This register is RAZ/WI when accessed as Non-secure.

This register is not banked between Security states.

## Field descriptions

The DAUTHCTRL bit assignments are:



### Bits [31:4]

Reserved, **RES0**.

### INTSPNIDEN, bit [3]

Internal Secure non-invasive debug enable. Overrides the external Secure non-invasive debug authentication interface.

The possible values of this bit are:

**0**

Secure Non-invasive debug prohibited.

**1**

Secure Non-invasive debug allowed.

Ignored if DAUTHCTRL.SPNIDENSEL == 0. See SecureNoninvasiveDebugEnabled().

This bit resets to zero on a Cold reset.

### SPNIDENSEL, bit [2]

Secure non-invasive debug enable select. Selects between DAUTHCTRL and the **IMPLEMENTATION DEFINED** external authentication interface for control of Secure non-invasive debug.

The possible values of this bit are:

**0**

Secure non-invasive debug controlled by the **IMPLEMENTATION DEFINED** external authentication interface. In the CoreSight authentication interface, this is controlled by the SPNIDEN signal.

**1**

Secure non-invasive debug controlled by DAUTHCTRL.INTSPNIDEN.

The PE ignores the value of this bit and Secure non-invasive debug is allowed if DHCSR.S\_SDE == 1. See SecureNoninvasiveDebugAllowed().

This bit resets to zero on a Cold reset.

**INTSPIDEN, bit [1]**

Internal Secure invasive debug enable. Overrides the external Secure invasive debug authentication interfaces.

The possible values of this bit are:

**0**

Secure halting and self-hosted debug prohibited.

**1**

Secure halting and self-hosted debug allowed.

Ignored if DAUTHCTRL.SPIDENSEL == 0. See SecureHaltingDebugAllowed() and SecureDebugMonitorAllowed().

This bit resets to zero on a Cold reset.

**SPIDENSEL, bit [0]**

Secure invasive debug enable select. Selects between DAUTHCTRL and the **IMPLEMENTATION DEFINED** external authentication interface for control of Secure invasive debug.

The possible values of this bit are:

**0**

Secure halting and self-hosted debug controlled by the **IMPLEMENTATION DEFINED** external authentication interface. In the CoreSight authentication interface, both are controlled by the SPIDEN signal.

**1**

Secure halting and self-hosted debug controlled by DAUTHCTRL.INTSPIDEN.

See SecureHaltingDebugAllowed() and SecureDebugMonitorAllowed().

This bit resets to zero on a Cold reset.



**0b00**

Security Extension not implemented.

**0b01**

Reserved.

**0b10**

Security Extension implemented and Secure invasive debug prohibited.

**0b11**

Security Extension implemented and Secure invasive debug allowed.

**NSNID, bits [3:2]**

Non-secure Non-invasive Debug. Indicates whether Non-secure non-invasive debug is allowed.

The possible values of this field are:

**0b0x**

Reserved.

**0b10**

Non-secure non-invasive debug prohibited.

**0b11**

Non-secure non-invasive debug allowed.

**NSID, bits [1:0]**

Non-secure Invasive Debug. Indicates whether Non-secure invasive debug is allowed.

The possible values of this field are:

**0b0x**

Reserved.

**0b10**

Non-secure invasive debug prohibited.

**0b11**

Non-secure invasive debug allowed.



### D1.2.21 DCCIMVAC, Data Cache line Clean and Invalidate by Address to PoC

The DCCIMVAC characteristics are:

#### Purpose

Clean and invalidate data or unified cache line by address to PoC.

#### Usage constraints

Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are [UNPREDICTABLE](#).

#### Configurations

This register is always implemented.

#### Attributes

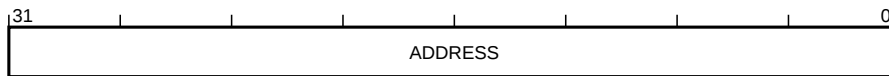
32-bit write-only register located at 0xE000EF70.

Secure software can access the Non-secure version of this register via DCCIMVAC\_NS located at 0xE002EF70. The location 0xE002EF70 is [RES0](#) to software executing in Non-secure state and the debugger.

This register is not banked between Security states.

#### Field descriptions

The DCCIMVAC bit assignments are:



#### ADDRESS, bits [31:0]

Address. Writing to this field initiates the maintenance operation for the address written.

## D1.2.22 DCCISW, Data Cache line Clean and Invalidate by Set/Way

The DCCISW characteristics are:

### Purpose

Clean and invalidate data or unified cache line by set/way.

### Usage constraints

Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are **UNPREDICTABLE**.

### Configurations

This register is always implemented.

### Attributes

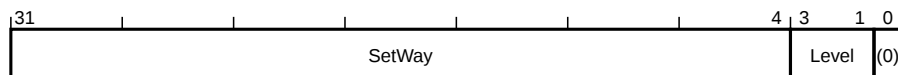
32-bit write-only register located at 0xE000EF74.

Secure software can access the Non-secure version of this register via DCCISW\_NS located at 0xE002EF74. The location 0xE002EF74 is **RES0** to software executing in Non-secure state and the debugger.

This register is not banked between Security states.

## Field descriptions

The DCCISW bit assignments are:



### SetWay, bits [31:4]

Cache set/way. Contains two fields: Way, bits[31:32-A], the number of the way to operate on. Set, bits[B-1:L], the number of the set to operate on. Bits[L-1:4] are **RES0**.  $A = \log_2(\text{ASSOCIATIVITY})$ ,  $L = \log_2(\text{LINELEN})$ ,  $B = (L + S)$ ,  $S = \log_2(\text{NSETS})$ . ASSOCIATIVITY, LINELEN (line length, in bytes), and NSETS (number of sets) have their usual meanings and are the values for the cache level being operated on. The values of A and S are rounded up to the next integer.

### Level, bits [3:1]

Cache level. Cache level to operate on, minus 1. For example, this field is 0 for operations on L1 cache, or 1 for operations on L2 cache.

### Bit [0]

Reserved, **RES0**.

### D1.2.23 DCCMVAC, Data Cache line Clean by Address to PoC

The DCCMVAC characteristics are:

**Purpose**

Clean data or unified cache line by address to PoC.

**Usage constraints**

Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are **UNPREDICTABLE**.

**Configurations**

This register is always implemented.

**Attributes**

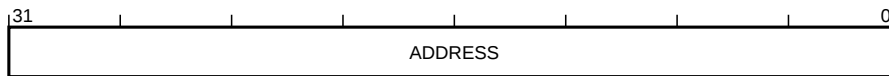
32-bit write-only register located at 0xE000EF68.

Secure software can access the Non-secure version of this register via DCCMVAC\_NS located at 0xE002EF68. The location 0xE002EF68 is **RES0** to software executing in Non-secure state and the debugger.

This register is not banked between Security states.

#### Field descriptions

The DCCMVAC bit assignments are:



**ADDRESS, bits [31:0]**

Address. Writing to this field initiates the maintenance operation for the address written.

### D1.2.24 DCCMVAU, Data Cache line Clean by address to PoU

The DCCMVAU characteristics are:

#### Purpose

Clean data or unified cache line by address to PoU.

#### Usage constraints

Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are [UNPREDICTABLE](#).

#### Configurations

This register is always implemented.

#### Attributes

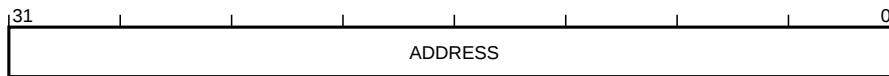
32-bit write-only register located at 0xE000EF64.

Secure software can access the Non-secure version of this register via DCCMVAU\_NS located at 0xE002EF64. The location 0xE002EF64 is [RES0](#) to software executing in Non-secure state and the debugger.

This register is not banked between Security states.

#### Field descriptions

The DCCMVAU bit assignments are:



#### ADDRESS, bits [31:0]

Address. Writing to this field initiates the maintenance operation for the address written.



## D1.2.26 DCIDR0, SCS Component Identification Register 0

The DCIDR0 characteristics are:

### Purpose

Provides CoreSight Unique Component Identifier information for the SCS.

### Usage constraints

Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are **UNPREDICTABLE**.

If the Main Extension is not implemented then it is **IMPLEMENTATION DEFINED** whether this register is accessible only to the debugger and **RES0** for software. Otherwise the register is accessible to the debugger and software.

### Configurations

Present only if CoreSight identification is implemented.

This register is **RES0** if CoreSight identification is not implemented.

Present only if Halting debug is implemented.

This register is **RES0** if Halting debug is not implemented.

### Attributes

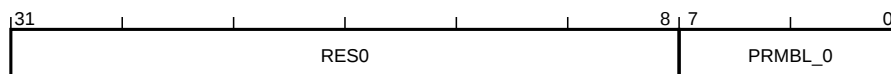
32-bit read-only register located at 0xE000EFF0.

Secure software can access the Non-secure version of this register via DCIDR0\_NS located at 0xE002EFF0. The location 0xE002EFF0 is **RES0** to software executing in Non-secure state and the debugger.

This register is not banked between Security states.

## Field descriptions

The DCIDR0 bit assignments are:



### Bits [31:8]

Reserved, **RES0**.

### PRMBL\_0, bits [7:0]

CoreSight component identification preamble. See the *Arm® CoreSight™ Architecture Specification*.

This field reads as 0x0D.



## D1.2.28 DCIDR2, SCS Component Identification Register 2

The DCIDR2 characteristics are:

### Purpose

Provides CoreSight Unique Component Identifier information for the SCS.

### Usage constraints

Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are **UNPREDICTABLE**.

If the Main Extension is not implemented then it is **IMPLEMENTATION DEFINED** whether this register is accessible only to the debugger and **RES0** for software. Otherwise the register is accessible to the debugger and software.

### Configurations

Present only if CoreSight identification is implemented.

This register is **RES0** if CoreSight identification is not implemented.

Present only if Halting debug is implemented.

This register is **RES0** if Halting debug is not implemented.

### Attributes

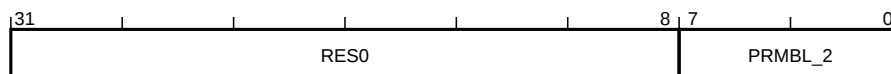
32-bit read-only register located at 0xE000EFF8.

Secure software can access the Non-secure version of this register via DCIDR2\_NS located at 0xE002EFF8. The location 0xE002EFF8 is **RES0** to software executing in Non-secure state and the debugger.

This register is not banked between Security states.

## Field descriptions

The DCIDR2 bit assignments are:



### Bits [31:8]

Reserved, **RES0**.

### PRMBL\_2, bits [7:0]

CoreSight component identification preamble. See the *Arm® CoreSight™ Architecture Specification*.

This field reads as 0x05.



### D1.2.29 DCIDR3, SCS Component Identification Register 3

The DCIDR3 characteristics are:

#### Purpose

Provides CoreSight Unique Component Identifier information for the SCS.

#### Usage constraints

Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are **UNPREDICTABLE**.

If the Main Extension is not implemented then it is **IMPLEMENTATION DEFINED** whether this register is accessible only to the debugger and **RES0** for software. Otherwise the register is accessible to the debugger and software.

#### Configurations

Present only if CoreSight identification is implemented.

This register is **RES0** if CoreSight identification is not implemented.

Present only if Halting debug is implemented.

This register is **RES0** if Halting debug is not implemented.

#### Attributes

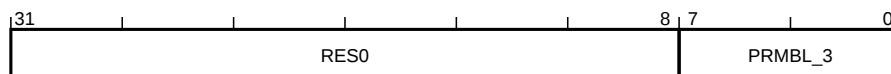
32-bit read-only register located at 0xE000EFFF.

Secure software can access the Non-secure version of this register via DCIDR3\_NS located at 0xE002EFFF. The location 0xE002EFFF is **RES0** to software executing in Non-secure state and the debugger.

This register is not banked between Security states.

### Field descriptions

The DCIDR3 bit assignments are:



#### Bits [31:8]

Reserved, **RES0**.

#### PRMBL\_3, bits [7:0]

CoreSight component identification preamble. See the *Arm® CoreSight™ Architecture Specification*.

This field reads as 0xB1.

### D1.2.30 DCIMVAC, Data Cache line Invalidate by Address to PoC

The DCIMVAC characteristics are:

#### Purpose

Invalidate data or unified cache line by address to PoC.

#### Usage constraints

Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are [UNPREDICTABLE](#).

#### Configurations

This register is always implemented.

#### Attributes

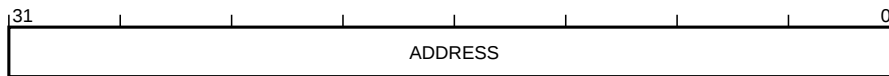
32-bit write-only register located at 0xE000EF5C.

Secure software can access the Non-secure version of this register via DCIMVAC\_NS located at 0xE002EF5C. The location 0xE002EF5C is [RES0](#) to software executing in Non-secure state and the debugger.

This register is not banked between Security states.

### Field descriptions

The DCIMVAC bit assignments are:



#### ADDRESS, bits [31:0]

Address. Writing to this field initiates the maintenance operation for the address written.

### D1.2.31 DCISW, Data Cache line Invalidate by Set/Way

The DCISW characteristics are:

#### Purpose

Invalidate data or unified cache line by set/way.

#### Usage constraints

Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

#### Configurations

This register is always implemented.

#### Attributes

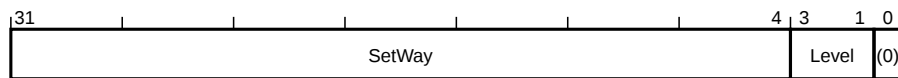
32-bit write-only register located at 0xE000EF60.

Secure software can access the Non-secure version of this register via DCISW\_NS located at 0xE002EF60. The location 0xE002EF60 is RES0 to software executing in Non-secure state and the debugger.

This register is not banked between Security states.

### Field descriptions

The DCISW bit assignments are:



#### SetWay, bits [31:4]

Cache set/way. Contains two fields: Way, bits[31:32-A], the number of the way to operate on. Set, bits[B-1:L], the number of the set to operate on. Bits[L-1:4] are RES0.  $A = \log_2(\text{ASSOCIATIVITY})$ ,  $L = \log_2(\text{LINELEN})$ ,  $B = (L + S)$ ,  $S = \log_2(\text{NSETS})$ . ASSOCIATIVITY, LINELEN (line length, in bytes), and NSETS (number of sets) have their usual meanings and are the values for the cache level being operated on. The values of A and S are rounded up to the next integer.

#### Level, bits [3:1]

Cache level. Cache level to operate on, minus 1. For example, this field is 0 for operations on L1 cache, or 1 for operations on L2 cache.

#### Bit [0]

Reserved, RES0.



### D1.2.33 DCRSR, Debug Core Register Select Register

The DCRSR characteristics are:

#### Purpose

With the DCRDR, provides debug access to the general-purpose registers, special-purpose registers, and the Floating-point Extension registers. A write to the DCRSR specifies the register to transfer, whether the transfer is a read or write, and starts the transfer.

#### Usage constraints

Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are **UNPREDICTABLE**.

Writes to this register while the PE is in Non-debug state are ignored.

This register is accessible only to the debugger and **RES0** to software.

#### Configurations

Present only if Halting debug is implemented.

This register is **RES0** if Halting debug is not implemented.

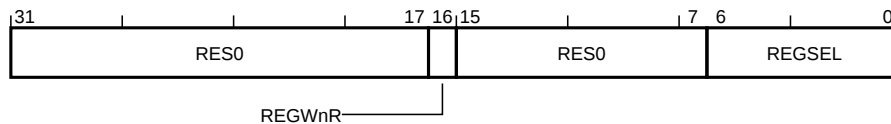
#### Attributes

32-bit write-only register located at 0xE000EDF4.

This register is not banked between Security states.

#### Field descriptions

The DCRSR bit assignments are:



#### Bits [31:17]

Reserved, **RES0**.

#### REGWnR, bit [16]

Register write/not-read. Specifies the access type for the transfer.

The possible values of this bit are:

**0**

Read.

**1**

Write.

#### Bits [15:7]

Reserved, **RES0**.

#### REGSEL, bits [6:0]

Register selector. Specifies the general-purpose register, special-purpose register, or Floating-point Extension register to transfer.

The possible values of this field are:

**0b0000000-0b0001100**

General-purpose registers R0-R12.

- 0b0001101**  
Current stack pointer, SP.
- 0b0001110**  
LR.
- 0b0001111**  
DebugReturnAddress.
- 0b0010000**  
XPSR.
- 0b0010001**  
Current state main stack pointer, SP\_main.
- 0b0010010**  
Current state process stack pointer, SP\_process.
- 0b0010100**  
Current state {CONTROL[7:0],FAULTMASK[7:0],BASEPRI[7:0],PRIMASK[7:0]}.  
If the Main Extension is not implemented, bits [23:8] of the transfer value are [RES0](#).
- 0b0011000**  
Non-secure main stack pointer, MSP\_NS.  
If the Security Extension is not implemented, this value is reserved.
- 0b0011001**  
Non-secure process stack pointer, PSP\_NS.  
If the Security Extension is not implemented, this value is reserved.
- 0b0011010**  
Secure main stack pointer, MSP\_S. Accessible only when DHCSR.S\_SDE == 1.  
If the Security Extension is not implemented, this value is reserved.
- 0b0011011**  
Secure process stack pointer, PSP\_S. Accessible only when DHCSR.S\_SDE == 1.  
If the Security Extension is not implemented, this value is reserved.
- 0b0011100**  
Secure main stack limit, MSPLIM\_S. Accessible only when DHCSR.S\_SDE == 1.  
If the Security Extension is not implemented, this value is reserved.
- 0b0011101**  
Secure process stack limit, PSPLIM\_S. Accessible only when DHCSR.S\_SDE == 1.  
If the Security Extension is not implemented, this value is reserved.
- 0b0011110**  
Non-secure main stack limit, MSPLIM\_NS.  
If the Main Extension is not implemented, this value is reserved.
- 0b0011111**  
Non-secure process stack limit, PSPLIM\_NS.  
If the Main Extension is not implemented, this value is reserved.
- 0b0100001**  
FPSCR.  
If the Floating-point Extension is not implemented, this value is reserved.

**0b0100010**

{CONTROL\_S[7:0],FAULTMASK\_S[7:0],BASEPRI\_S[7:0],PRIMASK\_S[7:0]}. Accessible only when DHCSR.S\_SDE == 1.

If the Main Extension is not implemented, bits [23:8] of the transfer value are **RES0**. If the Security Extension is not implemented, this value is reserved.

**0b0100011**

{CONTROL\_NS[7:0],FAULTMASK\_NS[7:0],BASEPRI\_NS[7:0],PRIMASK\_NS[7:0]}.

If the Main Extension is not implemented, bits [23:8] of the transfer value are **RES0**. If the Security Extension is not implemented, this value is reserved.

**0b1000000-0b1011111**

FP registers, S0-S31.

If the Floating-point Extension is not implemented, these values are reserved.

All other values are reserved.

If the Floating-point and Security Extensions are implemented, then FPSCR and S0-S31 are not accessible from Non-secure state if DHCSR.S\_SDE == 0 and either:

- FPCCR indicates the registers contain values from Secure state.
- NSACR prevents Non-secure access to the registers.

Registers that are not accessible are RAZ/WI.

If this field is written with a reserved value, the PE might behave as if a defined value was written, or ignore the value written, and the value of DCRDR becomes **UNKNOWN**.

## D1.2.34 DDEVARCH, SCS Device Architecture Register

The DDEVARCH characteristics are:

### Purpose

Provides CoreSight Unique Component Identifier information for the SCS.

### Usage constraints

Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are **UNPREDICTABLE**.

If the Main Extension is not implemented then it is **IMPLEMENTATION DEFINED** whether this register is accessible only to the debugger and **RES0** for software. Otherwise the register is accessible to the debugger and software.

### Configurations

Present only if CoreSight identification is implemented.

This register is **RES0** if CoreSight identification is not implemented.

Present only if Halting debug is implemented.

This register is **RES0** if Halting debug is not implemented.

### Attributes

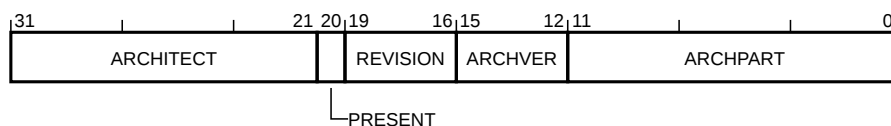
32-bit read-only register located at 0xE000EFBC.

Secure software can access the Non-secure version of this register via DDEVARCH\_NS located at 0xE002EFBC. The location 0xE002EFBC is **RES0** to software executing in Non-secure state and the debugger.

This register is not banked between Security states.

## Field descriptions

The DDEVARCH bit assignments are:



### ARCHITECT, bits [31:21]

Architect. Defines the architect of the component. Bits [31:28] are the JEP106 continuation code (JEP106 bank ID, minus 1) and bits [27:21] are the JEP106 ID code.

The possible values of this field are:

#### 0x23B

JEP106 continuation code 0x4, ID code 0x3B. Arm Limited.

Other values are defined by the JEDEC JEP106 standard.

This field reads as 0x23B.

### PRESENT, bit [20]

DEVARCH Present. Defines that the DEVARCH register is present.

The possible values of this bit are:

#### 1

DEVARCH information present.



This bit reads as one.

**REVISION, bits [19:16]**

Revision. Defines the architecture revision of the component.

The possible values of this field are:

**0b0000**

M-Profile debug architecture v3.0.

This field reads as 0b0000.

**ARCHVER, bits [15:12]**

Architecture Version. Defines the architecture version of the component.

The possible values of this field are:

**0b0010**

M-Profile debug architecture v3.

ARCHVER and ARCHPART are also defined as a single field, ARCHID, so that ARCHVER is ARCHID[15:12].

This field reads as 0b0010.

**ARCHPART, bits [11:0]**

Architecture Part. Defines the architecture of the component.

The possible values of this field are:

**0xA04**

M-Profile debug architecture.

ARCHVER and ARCHPART are also defined as a single field, ARCHID, so that ARCHPART is ARCHID[11:0].

This field reads as 0xA04.



## D1.2.36 DEMCR, Debug Exception and Monitor Control Register

The DEMCR characteristics are:

### Purpose

Manages vector catch behavior and DebugMonitor handling when debugging.

### Usage constraints

Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are **UNPREDICTABLE**.

If the Main Extension is not implemented then it is **IMPLEMENTATION DEFINED** whether this register is accessible only to the debugger and **RES0** for software. Otherwise the register is accessible to the debugger and software.

### Configurations

Present if Halting debug or the Main Extension is implemented.

This register is **RES0** if both Halting debug and Main Extension are not implemented.

### Attributes

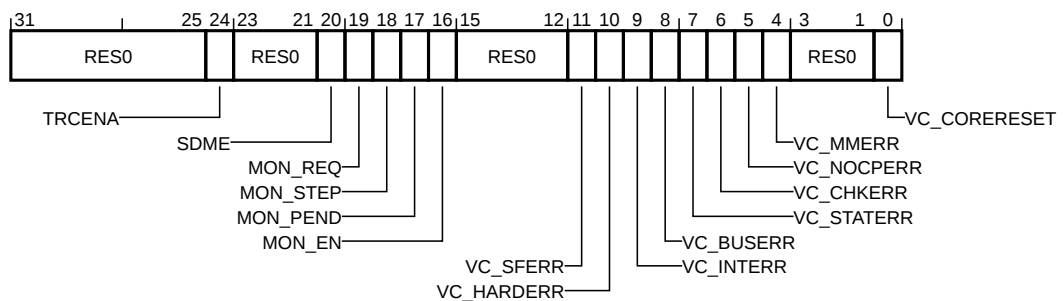
32-bit read/write register located at 0xE000EDFC.

Secure software can access the Non-secure version of this register via DEMCR\_NS located at 0xE002EDFC. The location 0xE002EDFC is **RES0** to software executing in Non-secure state and the debugger.

This register is not banked between Security states.

## Field descriptions

The DEMCR bit assignments are:



### Bits [31:25]

Reserved, **RES0**.

### TRCENA, bit [24]

Trace enable. Global enable for all DWT and ITM features.

The possible values of this bit are:

**0**

DWT and ITM features disabled.

**1**

DWT and ITM features enabled.

If the DWT and ITM units are not implemented, this bit is **RES0**. See the descriptions of DWT and ITM for details of which features this bit controls.

Setting this bit to 0 might not stop all events. To ensure that all events are stopped, software must set all DWT and ITM feature enable bits to 0, and ensure that all trace generated by the DWT and ITM has been flushed, before setting this bit to 0.

It is **IMPLEMENTATION DEFINED** whether this bit affects how the system processes trace.

Arm recommends that this bit is set to 1 when using an ETM even if any implemented DWT and ITM are not being used.

This bit resets to zero on a Cold reset.

**Bits [23:21]**

Reserved, **RES0**.

**SDME, bit [20]**

Secure DebugMonitor enable. Indicates whether the DebugMonitor targets the Secure or the Non-secure state and whether debug events are allowed in Secure state.

The possible values of this bit are:

**0**

Debug events prohibited in Secure state and the DebugMonitor exception targets Non-secure state.

**1**

Debug events allowed in Secure state and the DebugMonitor exception targets Secure state.

When DebugMonitor exception is not pending or active, this bit reflects the value of SecureDebugMonitorAllowed(), otherwise, the previous value is retained.

This bit is read-only.

If the Security Extension is not implemented, this bit is **RES0**.

If the Main Extension is not implemented, this bit is **RES0**.

**MON\_REQ, bit [19]**

Monitor request. DebugMonitor semaphore bit.

The PE does not use this bit. The monitor software defines the meaning and use of this bit.

If the Main Extension is not implemented, this bit is **RES0**.

This bit resets to zero on a Warm reset.

**MON\_STEP, bit [18]**

Monitor step. Enable DebugMonitor exception stepping.

The possible values of this bit are:

**0**

Stepping disabled.

**1**

Stepping enabled.

The effect of changing this bit at an execution priority that is lower than the priority of the DebugMonitor exception is **UNPREDICTABLE**.

The effect of a write to DEMCR.MON\_STEP is **CONSTRAINED UNPREDICTABLE** unless one of:

- The PE is executing at a priority that is greater than or the same as the priority of the DebugMonitor exception.
- Before the write, DEMCR.MON\_STEP == 0 and the write writes 0 to DEMCR.MON\_STEP.

The **CONSTRAINED UNPREDICTABLE** write does one of the following:

- The write is ignored.

- DEMCR.MON\_STEP is set to 1. The effect of the subsequent instruction is **UNPREDICTABLE**, and the PE might generate a DebugMonitor exception at any time, if enabled and permitted.
- DEMCR.MON\_STEP is set to 0. It is **UNPREDICTABLE** whether an 'in progress' step completes.

If the Main Extension is not implemented, this bit is **RES0**.

This bit resets to zero on a Warm reset.

#### **MON\_PEND, bit [17]**

Monitor pend. Sets or clears the pending state of the DebugMonitor exception.

The possible values of this bit are:

**0**

Clear the status of the DebugMonitor exception to not pending.

**1**

Set the status of the DebugMonitor exception to pending.

When the DebugMonitor exception is pending it becomes active subject to the exception priority rules. The effect of setting this bit to 1 is not affected by the value of the MON\_EN bit. This means that software or a debugger can set MON\_PEND to 1 and pend a DebugMonitor exception, even when MON\_EN is set to 0.

If the Main Extension is not implemented, this bit is **RES0**.

This bit resets to zero on a Warm reset.

#### **MON\_EN, bit [16]**

Monitor enable. Enable the DebugMonitor exception.

The possible values of this bit are:

**0**

DebugMonitor exception disabled.

**1**

DebugMonitor exception enabled.

If a debug event halts the PE, the PE ignores the value of this bit.

If DEMCR.SDME is one this bit is RAZ/WI from Non-secure state

If the Main Extension is not implemented, this bit is **RES0**.

This bit resets to zero on a Warm reset.

#### **Bits [15:12]**

Reserved, **RES0**.

#### **VC\_SFERR, bit [11]**

Vector Catch SecureFault. SecureFault exception Halting debug vector catch enable.

The possible values of this bit are:

**0**

Halting debug trap on SecureFault disabled.

**1**

Halting debug trap on SecureFault enabled.

The PE ignores the value of this bit if DHCSR.C\_DEBUGEN == 0, HaltingDebugAllowed() == FALSE, or DHCSR.S\_SDE == 0.

If the Security Extension is not implemented, this bit is **RES0**.

If the Main Extension is not implemented, this bit is **RES0**.

If Halting debug is not implemented, this bit is [RES0](#).

This bit resets to zero on a Cold reset.

#### **VC\_HARDERR, bit [10]**

Vector Catch HardFault errors. HardFault exception Halting debug vector catch enable.

The possible values of this bit are:

**0**

Halting debug trap on HardFault disabled.

**1**

Halting debug trap on HardFault enabled.

The PE ignores the value of this bit if `DHCSR.C_DEBUGEN == 0`, `HaltingDebugAllowed() == FALSE`, or the Security Extension is implemented, `DHCSR.S_SDE == 0` and the exception targets Secure state.

If Halting debug is not implemented, this bit is [RES0](#).

This bit resets to zero on a Cold reset.

#### **VC\_INTERR, bit [9]**

Vector Catch interrupt errors. Enable Halting debug vector catch for faults arising from lazy state preservation, stack violations and context stacking or unstacking during exception entry or return.

The possible values of this bit are:

**0**

Halting debug trap on faults disabled.

**1**

Halting debug trap on faults enabled.

The PE ignores the value of this bit if `DHCSR.C_DEBUGEN == 0`, `HaltingDebugAllowed() == FALSE`, or the Security Extension is implemented, `DHCSR.S_SDE == 0` and the exception targets Secure state.

If the Main Extension is not implemented, this bit is [RES0](#).

If Halting debug is not implemented, this bit is [RES0](#).

This bit resets to zero on a Cold reset.

#### **VC\_BUSERR, bit [8]**

Vector Catch BusFault errors. BusFault exception Halting debug vector catch enable.

The possible values of this bit are:

**0**

Halting debug trap on BusFault disabled.

**1**

Halting debug trap on BusFault enabled.

The PE ignores the value of this bit if `DHCSR.C_DEBUGEN == 0`, `HaltingDebugAllowed() == FALSE`, or the Security Extension is implemented, `DHCSR.S_SDE == 0` and the exception targets Secure state.

If the Main Extension is not implemented, this bit is [RES0](#).

If Halting debug is not implemented, this bit is [RES0](#).

This bit resets to zero on a Cold reset.

#### **VC\_STATERR, bit [7]**

Vector Catch state errors. Enable Halting debug trap on a UsageFault exception caused by a state information error, for example an Undefined Instruction exception.

The possible values of this bit are:

**0**  
Halting debug trap on UsageFault caused by state information error disabled.

**1**  
Halting debug trap on UsageFault caused by state information error enabled.

The PE ignores the value of this bit if `DHCSR.C_DEBUGEN == 0`, `HaltingDebugAllowed() == FALSE`, or the Security Extension is implemented, `DHCSR.S_SDE == 0` and the exception targets Secure state.

If the Main Extension is not implemented, this bit is [RES0](#).

If Halting debug is not implemented, this bit is [RES0](#).

This bit resets to zero on a Cold reset.

#### **VC\_CHKERR, bit [6]**

Vector Catch check errors. Enable Halting debug trap on a UsageFault exception caused by an alignment check error or divide-by-zero trap.

The possible values of this bit are:

**0**  
Halting debug trap on UsageFault caused by checking error disabled.

**1**  
Halting debug trap on UsageFault caused by checking error enabled.

The PE ignores the value of this bit if `DHCSR.C_DEBUGEN == 0`, `HaltingDebugAllowed() == FALSE`, or the Security Extension is implemented, `DHCSR.S_SDE == 0` and the exception targets Secure state.

If the Main Extension is not implemented, this bit is [RES0](#).

If Halting debug is not implemented, this bit is [RES0](#).

This bit resets to zero on a Cold reset.

#### **VC\_NOCPERR, bit [5]**

Vector Catch NOCP errors. Enable Halting debug trap on a UsageFault caused by an access to a coprocessor.

The possible values of this bit are:

**0**  
Halting debug trap on UsageFault caused by access to a coprocessor disabled.

**1**  
Halting debug trap on UsageFault caused by access to a coprocessor enabled.

The PE ignores the value of this bit if `DHCSR.C_DEBUGEN == 0`, `HaltingDebugAllowed() == FALSE`, or the Security Extension is implemented, `DHCSR.S_SDE == 0` and the exception targets Secure state.

If the Main Extension is not implemented, this bit is [RES0](#).

If Halting debug is not implemented, this bit is [RES0](#).

This bit resets to zero on a Cold reset.

#### **VC\_MMERR, bit [4]**

Vector Catch MemManage errors. Enable Halting debug trap on a MemManage exception.

The possible values of this bit are:

**0**  
Halting debug trap on MemManage disabled.

**1**  
Halting debug trap on MemManage enabled.

The PE ignores the value of this bit if `DHCSR.C_DEBUGEN == 0`, `HaltingDebugAllowed() == FALSE`, or the Security Extension is implemented, `DHCSR.S_SDE == 0` and the exception targets Secure state.

If the Main Extension is not implemented, this bit is [RES0](#).

If Halting debug is not implemented, this bit is [RES0](#).

This bit resets to zero on a Cold reset.

**Bits [3:1]**

Reserved, [RES0](#).

**VC\_CORERESET, bit [0]**

Vector Catch Core reset. Enable Reset Vector Catch. This causes a Warm reset to halt a running system.

The possible values of this bit are:

**0**

Halting debug trap on reset disabled.

**1**

Halting debug trap on reset enabled.

If `DHCSR.C_DEBUGEN == 0` or `HaltingDebugAllowed() == FALSE`, the PE ignores the value of this bit. Otherwise, when this bit is set to 1 a Warm reset will pend a Vector Catch debug event. The debug event is pended even the PE resets into Secure state and `DHCSR.S_SDE == 0`.

If Halting debug is not implemented, this bit is [RES0](#).

This bit resets to zero on a Cold reset.



### D1.2.37 DFSR, Debug Fault Status Register

The DFSR characteristics are:

**Purpose**

Shows which debug event occurred.

**Usage constraints**

Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

If the Main Extension is not implemented then it is IMPLEMENTATION DEFINED whether this register is accessible only to the debugger and RES0 for software. Otherwise the register is accessible to the debugger and software.

**Configurations**

Present if Halting debug or the Main Extension is implemented.

This register is RES0 if both Halting debug and Main Extension are not implemented.

**Attributes**

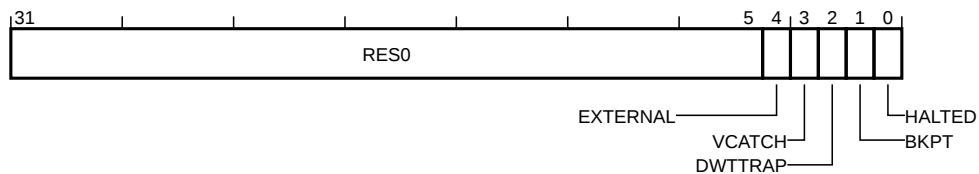
32-bit read/write-one-to-clear register located at 0xE000ED30.

Secure software can access the Non-secure version of this register via DFSR\_NS located at 0xE002ED30. The location 0xE002ED30 is RES0 to software executing in Non-secure state and the debugger.

This register is not banked between Security states.

### Field descriptions

The DFSR bit assignments are:



**Bits [31:5]**

Reserved, RES0.

**EXTERNAL, bit [4]**

External event. Sticky flag indicating whether an External debug request debug event has occurred.

The possible values of this bit are:

**0**  
Debug event has not occurred.

**1**  
Debug event has occurred.

This bit resets to zero on a Cold reset.

**VCATCH, bit [3]**

Vector Catch event. Sticky flag indicating whether a Vector catch debug event has occurred.

The possible values of this bit are:

**0**  
Debug event has not occurred.

**1**

Debug event has occurred.

If Halting debug is not implemented, this bit is [RES0](#).

This bit resets to zero on a Cold reset.

**DWTTRAP, bit [2]**

Watchpoint event. Sticky flag indicating whether a Watchpoint debug event has occurred.

The possible values of this bit are:

**0**

Debug event has not occurred.

**1**

Debug event has occurred.

If the DWT is not implemented, this bit is [RES0](#).

This bit resets to zero on a Cold reset.

**BKPT, bit [1]**

Breakpoint event. Sticky flag indicating whether a Breakpoint debug event has occurred.

The possible values of this bit are:

**0**

Debug event has not occurred.

**1**

Debug event has occurred.

This bit resets to zero on a Cold reset.

**HALTED, bit [0]**

Halt or step event. Sticky flag indicating that a Halt request debug event or Step debug event has occurred.

The possible values of this bit are:

**0**

Debug event has not occurred.

**1**

Debug event has occurred.

This bit resets to zero on a Cold reset.

### D1.2.38 DHCSR, Debug Halting Control and Status Register

The DHCSR characteristics are:

**Purpose**

Controls Halting debug.

**Usage constraints**

Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are **UNPREDICTABLE**.

It is **IMPLEMENTATION DEFINED** whether this register is accessible only to the debugger and **RES0** for software.

**Configurations**

Present only if Halting debug is implemented.

This register is **RES0** if Halting debug is not implemented.

**Attributes**

32-bit read/write register located at 0xE00EDF0.

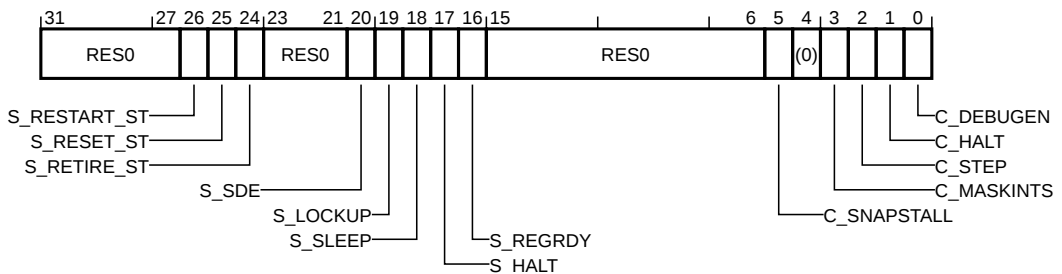
Secure software can access the Non-secure version of this register via DHCSR\_NS located at 0xE002EDF0. The location 0xE002EDF0 is **RES0** to software executing in Non-secure state and the debugger.

This register is not banked between Security states.

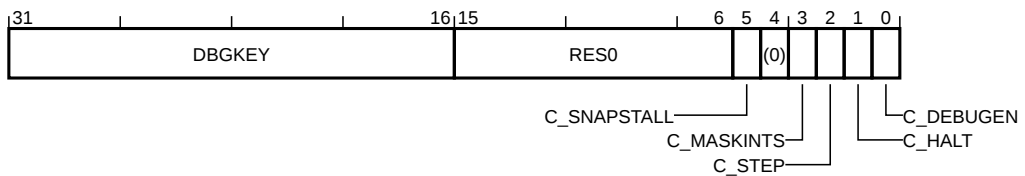
#### Field descriptions

The DHCSR bit assignments are:

On a read:



On a write:



**DBGKEY, bits [31:16], on a write**

Debug key. A debugger must write 0xA05F to this field to enable write access to the remaining bits, otherwise the PE ignores the write access.

The possible values of this field are:

**0xA05F**

Writes accompanied by this value update bits[15:0].

**Not 0xA05F**

Write ignored.

**Bits [31:27], on a read**

Reserved, **RES0**.

**S\_RESTART\_ST, bit [26], on a read**

Restart sticky status. Indicates the PE has processed a request to clear DHCSR.C\_HALT to 0. That is, either a write to DHCSR that clears DHCSR.C\_HALT from 1 to 0, or an External Restart Request.

The possible values of this bit are:

**0**

PE has not left Debug state since the last read of DHCSR.

**1**

PE has left Debug state since the last read of DHCSR.

If the PE is not halted when C\_HALT is cleared to zero, it is **UNPREDICTABLE** whether this bit is set to 1. If DHCSR.C\_DEBUGEN == 0 this bit reads as an **UNKNOWN** value.

This bit clears to zero when read.

**Note**

If the request to clear C\_HALT is made simultaneously with a request to set C\_HALT, for example a restart request and external debug request occur together, then the PE notionally leaves Debug state and immediately halts again and S\_RESTART\_ST is set to 1.

**S\_RESET\_ST, bit [25], on a read**

Reset sticky status. Indicates whether the PE has been reset since the last read of the DHCSR.

The possible values of this bit are:

**0**

No reset since last DHCSR read.

**1**

At least one reset since last DHCSR read.

This bit clears to zero when read.

This bit resets to one on a Warm reset.

**S\_RETIRE\_ST, bit [24], on a read**

Retire sticky status. Set to 1 every time the PE retires one or more instructions.

The possible values of this bit are:

**0**

No instruction retired since last DHCSR read.

**1**

At least one instruction retired since last DHCSR read.

This bit clears to zero when read.

This bit resets to an **UNKNOWN** value on a Warm reset.

**Bits [23:21], on a read**

Reserved, **RES0**.

**S\_SDE, bit [20], on a read**

Secure debug enabled. Indicates whether Secure invasive debug is allowed.

The possible values of this bit are:

**0**  
Secure invasive debug prohibited.

**1**  
Secure invasive debug allowed.

If the PE is in Non-debug state, this bit reflects the value of SecureHaltingDebugEnabled().

If the PE is in Debug state then this bit is 1 if the PE entered Debug state from either Non-secure state with SecureHaltingDebugEnabled() == TRUE or from Secure state, and 0 otherwise. The value of this bit does not change while the PE remains in Debug state.

If the Security Extension is not implemented, this bit is [RES0](#).

**S\_LOCKUP, bit [19], on a read**

Lockup status. Indicates whether the PE is in Lockup state.

The possible values of this bit are:

**0**  
Not locked up.

**1**  
Locked up.

This bit can only be read as 1 by a remote debugger, using the DAP. The value of 1 indicates that the PE is running but locked up. The bit clears to 0 when the PE enters Debug state.

**S\_SLEEP, bit [18], on a read**

Sleeping status. Indicates whether the PE is sleeping.

The possible values of this bit are:

**0**  
Not sleeping.

**1**  
Sleeping.

The debugger must set the C\_HALT bit to 1 to gain control, or wait for an interrupt or other wakeup event to wakeup the system.

**S\_HALT, bit [17], on a read**

Halted status. Indicates whether the PE is in Debug state.

The possible values of this bit are:

**0**  
In Non-debug state.

**1**  
In Debug state.

**S\_REGRDY, bit [16], on a read**

Register ready status. Handshake flag to transfers through the DCRDR.

The possible values of this bit are:

**0**  
Write to DCRSR performed, but transfer not yet complete.

**1**  
Transfer complete, or no outstanding transfer.

This bit is valid only when the PE is in Debug state, otherwise this bit is [UNKNOWN](#).

This bit resets to an [UNKNOWN](#) value on a Warm reset.

**Bits [15:6]**

Reserved, [RES0](#).

**C\_SNAPSTALL, bit [5]**

Snap stall control. Allow imprecise entry to Debug state.

The possible values of this bit are:

**0**

No action.

**1**

Allows imprecise entry to Debug state, for example by forcing any stalled load or store instruction to be abandoned.

Setting this bit to 1 allows a debugger to request an imprecise entry to Debug state. Writing 1 to this bit makes the state of the memory system [UNPREDICTABLE](#). Therefore if a debugger writes 1 to this bit it must reset the system before leaving Debug state.

The effect of setting this bit to 1 is [UNPREDICTABLE](#) unless the DHCSR write also sets C\_DEBUGEN and C\_HALT to 1. This means that if the PE is not already in Debug state, it enters Debug state when the stalled instruction completes.

If the Security Extension is implemented, then writes to this bit are ignored when DHCSR.S\_SDE == 0.

If DHCSR.C\_DEBUGEN == 0 or HaltingDebugAllowed() == FALSE, the PE ignores this bit and behaves as if it is set to 0.

If the Main Extension is not implemented, this bit is [RES0](#).

This bit resets to zero on a Cold reset.

**Note**

A debugger can write to the DHCSR to clear this bit to 0. However, this does not remove the [UNPREDICTABLE](#) state of the memory system caused by setting C\_SNAPSTALL to 1. The architecture does not guarantee that setting this bit to 1 will force an entry to Debug state. Arm strongly recommends that a value of 1 is never written to C\_SNAPSTALL when the PE is in Debug state.

**Bit [4]**

Reserved, [RES0](#).

**C\_MASKINTS, bit [3]**

Mask interrupts control. When debug is enabled, the debugger can write to this bit to mask PendSV, SysTick and external configurable interrupts.

The possible values of this bit are:

**0**

Do not mask.

**1**

Mask PendSV, SysTick and external configurable interrupts.

The effect of any single write to DHCSR that changes the value of this bit is [UNPREDICTABLE](#) unless one of:

- Before the write, DHCSR.{S\_HALT,C\_HALT} are both set to 1 and the write also writes 1 to DHCSR.C\_HALT.
- Before the write, DHCSR.C\_DEBUGEN == 0 or HaltingDebugAllowed() == FALSE, and the write writes 0 to DHCSR.C\_MASKINTS.

This means that a single write to DHCSR must not clear DHCSR.C\_HALT to 0 and change the value of the C\_MASKINTS bit.

If the Security Extension is implemented and  $\text{DHCSR.S\_SDE} == 0$ , this bit does not affect interrupts targeting Secure state.

If  $\text{DHCSR.C\_DEBUGEN} == 0$  or  $\text{HaltingDebugAllowed()} == \text{FALSE}$ , the PE ignores this bit and behaves as if it is set to 0.

If  $\text{DHCSR.C\_DEBUGEN} == 0$  this bit reads as an **UNKNOWN** value.

This bit resets to an **UNKNOWN** value on a Cold reset.

**Note**

This bit does not affect NMI.

**C\_STEP, bit [2]**

Step control. Enable single instruction step.

The possible values of this bit are:

**0**  
No effect.

**1**  
Single step enabled.

The effect of a single write to DHCSR that changes the value of this bit is **UNPREDICTABLE** unless one of:

- Before the write,  $\text{DHCSR}\{S\_HALT, C\_HALT\}$  are both set to 1.
- Before the write,  $\text{DHCSR.C\_DEBUGEN} == 0$  or  $\text{HaltingDebugAllowed()} == \text{FALSE}$  or  $\text{DHCSR.C\_STEP} == 0$ , and the write writes 0 to  $\text{DHCSR.C\_STEP}$ .

The PE ignores this bit and behaves as if it set to 0 if any of:

- $\text{DHCSR.C\_DEBUGEN} == 0$  or  $\text{HaltingDebugAllowed()} == \text{FALSE}$ .
- The Security Extension is implemented,  $\text{DHCSR.S\_SDE} == 0$  and the PE is in Secure state.

If  $\text{DHCSR.C\_DEBUGEN} == 0$  this bit reads as an **UNKNOWN** value.

This bit resets to an **UNKNOWN** value on a Cold reset.

**C\_HALT, bit [1]**

Halt control. PE to enter Debug state halt request.

The possible values of this bit are:

**0**  
Causes the PE to leave Debug state, if in Debug state.

**1**  
Halt the PE.

The PE sets  $\text{C\_HALT}$  to 1 when a debug event pends an entry to Debug state.

The PE ignores this bit and behaves as if it is set to 0 if any of:

- $\text{DHCSR.C\_DEBUGEN} == 0$  or  $\text{HaltingDebugAllowed()} == \text{FALSE}$ .
- The Security Extension is implemented,  $\text{DHCSR.S\_SDE} == 0$  and the PE is in Secure state.

If  $\text{DHCSR.C\_DEBUGEN} == 0$  this bit reads as an **UNKNOWN** value.

This bit resets to zero on a Warm reset.

**C\_DEBUGEN, bit [0]**

Debug enable control. Enable Halting debug.

The possible values of this bit are:

**0**  
Disabled.

**1**  
Enabled.

If a debugger writes to DHCSR to change the value of this bit from 0 to 1, it must also write 0 to the C\_MASKINTS bit, otherwise behavior is **UNPREDICTABLE**.

If this bit is set to 0:

- The PE behaves as if DHCSR.{C\_MASKINTS, C\_STEP, C\_HALT} are all set to 0.
- DHCSR.{S\_RESTART\_ST, C\_MASKINTS, C\_STEP, C\_HALT} are **UNKNOWN** on reads of DHCSR.

This bit is read/write to the debugger. Writes from software are ignored.

This bit resets to zero on a Cold reset.



### D1.2.39 DLAR, SCS Software Lock Access Register

The DLAR characteristics are:

#### Purpose

Provides CoreSight Software Lock control for the SCS, see the *Arm® CoreSight™ Architecture Specification* for details.

#### Usage constraints

Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are **UNPREDICTABLE**.

This register is RAZ/WI if accessed via the debugger.

#### Configurations

Present only if CoreSight identification is implemented.

This register is **RES0** if CoreSight identification is not implemented.

Present only if the optional Software Lock is implemented.

This register is RAZ/WI if the Software Lock is not implemented.

#### Attributes

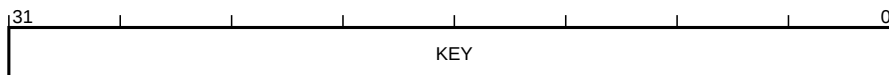
32-bit write-only register located at 0xE000EFB0.

Secure software can access the Non-secure version of this register via DLAR\_NS located at 0xE002EFB0. The location 0xE002EFB0 is **RES0** to software executing in Non-secure state and the debugger.

This register is not banked between Security states.

#### Field descriptions

The DLAR bit assignments are:



#### KEY, bits [31:0]

Lock Access control.

Writing the key value 0xC5ACCE55 to this field unlocks the lock, enabling write accesses to the registers of this component through a memory-mapped interface.

Writing any other value to this register locks the lock, disabling write accesses to the registers of this component through a memory mapped interface.

## D1.2.40 DLSR, SCS Software Lock Status Register

The DLSR characteristics are:

### Purpose

Provides CoreSight Software Lock status information for the SCS, see the *Arm® CoreSight™ Architecture Specification* for details.

### Usage constraints

Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are **UNPREDICTABLE**.

This register is RAZ/WI if accessed via the debugger.

### Configurations

Present only if CoreSight identification is implemented.

This register is **RES0** if CoreSight identification is not implemented.

Present only if the optional Software Lock is implemented.

This register is RAZ/WI if the Software Lock is not implemented.

### Attributes

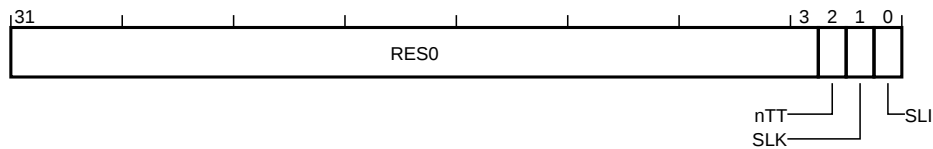
32-bit read-only register located at 0xE000EFB4.

Secure software can access the Non-secure version of this register via DLSR\_NS located at 0xE002EFB4. The location 0xE002EFB4 is **RES0** to software executing in Non-secure state and the debugger.

This register is not banked between Security states.

## Field descriptions

The DLSR bit assignments are:



### Bits [31:3]

Reserved, **RES0**.

### nTT, bit [2]

Not thirty-two bit. See the *Arm® CoreSight™ Architecture Specification*.

This bit reads as zero.

### SLK, bit [1]

Software Lock status. See the *Arm® CoreSight™ Architecture Specification*.

The possible values of this bit are:

**0**

Lock clear. Software writes are permitted to the registers of the component.

**1**

Lock set. Software writes to the registers of this component are ignored, and reads have no side-effects.

For a debugger read of this register, or when the Software Lock is not implemented, this bit is **RES0**.

This bit resets to one on a Warm reset.

**SLI, bit [0]**

Software Lock implemented. See the *Arm® CoreSight™ Architecture Specification*.

The possible values of this bit are:

**0**

Software Lock not implemented or debugger access.

**1**

Software Lock is implemented and software access.

For a debugger read of this register, or when the Software Lock is not implemented, this bit is RAZ.

This bit reads as an **IMPLEMENTATION DEFINED** value.



## D1.2.42 DPIDR1, SCS Peripheral Identification Register 1

The DPIDR1 characteristics are:

### Purpose

Provides CoreSight Unique Component Identifier information for the SCS.

### Usage constraints

Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are **UNPREDICTABLE**.

If the Main Extension is not implemented then it is **IMPLEMENTATION DEFINED** whether this register is accessible only to the debugger and **RES0** for software. Otherwise the register is accessible to the debugger and software.

### Configurations

Present only if CoreSight identification is implemented.

This register is **RES0** if CoreSight identification is not implemented.

Present only if Halting debug is implemented.

This register is **RES0** if Halting debug is not implemented.

### Attributes

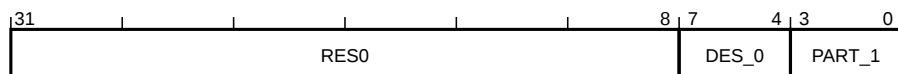
32-bit read-only register located at 0xE000EFE4.

Secure software can access the Non-secure version of this register via DPIDR1\_NS located at 0xE002EFE4. The location 0xE002EFE4 is **RES0** to software executing in Non-secure state and the debugger.

This register is not banked between Security states.

## Field descriptions

The DPIDR1 bit assignments are:



### Bits [31:8]

Reserved, **RES0**.

### DES\_0, bits [7:4]

JEP106 identification code bits [3:0]. See the *Arm® CoreSight™ Architecture Specification*.

This field reads as an **IMPLEMENTATION DEFINED** value.

### PART\_1, bits [3:0]

Part number bits [11:8]. See the *Arm® CoreSight™ Architecture Specification*.

This field reads as an **IMPLEMENTATION DEFINED** value.

### D1.2.43 DPIDR2, SCS Peripheral Identification Register 2

The DPIDR2 characteristics are:

#### Purpose

Provides CoreSight Unique Component Identifier information for the SCS.

#### Usage constraints

Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are **UNPREDICTABLE**.

If the Main Extension is not implemented then it is **IMPLEMENTATION DEFINED** whether this register is accessible only to the debugger and **RES0** for software. Otherwise the register is accessible to the debugger and software.

#### Configurations

Present only if CoreSight identification is implemented.

This register is **RES0** if CoreSight identification is not implemented.

Present only if Halting debug is implemented.

This register is **RES0** if Halting debug is not implemented.

#### Attributes

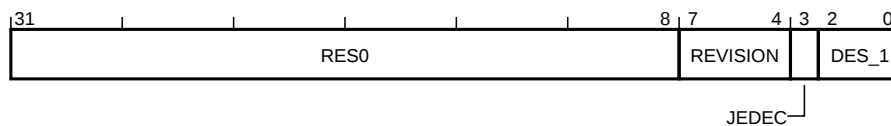
32-bit read-only register located at 0xE000EFE8.

Secure software can access the Non-secure version of this register via DPIDR2\_NS located at 0xE002EFE8. The location 0xE002EFE8 is **RES0** to software executing in Non-secure state and the debugger.

This register is not banked between Security states.

### Field descriptions

The DPIDR2 bit assignments are:



#### Bits [31:8]

Reserved, **RES0**.

#### REVISION, bits [7:4]

Component revision. See the *Arm® CoreSight™ Architecture Specification*.

This field reads as an **IMPLEMENTATION DEFINED** value.

#### JEDEC, bit [3]

JEDEC assignee value is used. See the *Arm® CoreSight™ Architecture Specification*.

This bit reads as one.

#### DES\_1, bits [2:0]

JEP106 identification code bits [6:4]. See the *Arm® CoreSight™ Architecture Specification*.

This field reads as an **IMPLEMENTATION DEFINED** value.







## D1.2.46 DPIDR5, SCS Peripheral Identification Register 5

The DPIDR5 characteristics are:

### Purpose

Provides CoreSight Unique Component Identifier information for the SCS.

### Usage constraints

Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are **UNPREDICTABLE**.

If the Main Extension is not implemented then it is **IMPLEMENTATION DEFINED** whether this register is accessible only to the debugger and **RES0** for software. Otherwise the register is accessible to the debugger and software.

### Configurations

Present only if CoreSight identification is implemented.

This register is **RES0** if CoreSight identification is not implemented.

Present only if Halting debug is implemented.

This register is **RES0** if Halting debug is not implemented.

### Attributes

32-bit read-only register located at 0xE000EFD4.

Secure software can access the Non-secure version of this register via DPIDR5\_NS located at 0xE002EFD4. The location 0xE002EFD4 is **RES0** to software executing in Non-secure state and the debugger.

This register is not banked between Security states.

## Field descriptions

The DPIDR5 bit assignments are:



### Bits [31:0]

Reserved, **RES0**.

### D1.2.47 DPIDR6, SCS Peripheral Identification Register 6

The DPIDR6 characteristics are:

#### Purpose

Provides CoreSight Unique Component Identifier information for the SCS.

#### Usage constraints

Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are **UNPREDICTABLE**.

If the Main Extension is not implemented then it is **IMPLEMENTATION DEFINED** whether this register is accessible only to the debugger and **RES0** for software. Otherwise the register is accessible to the debugger and software.

#### Configurations

Present only if CoreSight identification is implemented.

This register is **RES0** if CoreSight identification is not implemented.

Present only if Halting debug is implemented.

This register is **RES0** if Halting debug is not implemented.

#### Attributes

32-bit read-only register located at 0xE000EFD8.

Secure software can access the Non-secure version of this register via DPIDR6\_NS located at 0xE002EFD8. The location 0xE002EFD8 is **RES0** to software executing in Non-secure state and the debugger.

This register is not banked between Security states.

### Field descriptions

The DPIDR6 bit assignments are:



#### Bits [31:0]

Reserved, **RES0**.

## D1.2.48 DPIDR7, SCS Peripheral Identification Register 7

The DPIDR7 characteristics are:

### Purpose

Provides CoreSight Unique Component Identifier information for the SCS.

### Usage constraints

Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are **UNPREDICTABLE**.

If the Main Extension is not implemented then it is **IMPLEMENTATION DEFINED** whether this register is accessible only to the debugger and **RES0** for software. Otherwise the register is accessible to the debugger and software.

### Configurations

Present only if CoreSight identification is implemented.

This register is **RES0** if CoreSight identification is not implemented.

Present only if Halting debug is implemented.

This register is **RES0** if Halting debug is not implemented.

### Attributes

32-bit read-only register located at 0xE000EFDC.

Secure software can access the Non-secure version of this register via DPIDR7\_NS located at 0xE002EFDC. The location 0xE002EFDC is **RES0** to software executing in Non-secure state and the debugger.

This register is not banked between Security states.

## Field descriptions

The DPIDR7 bit assignments are:



### Bits [31:0]

Reserved, **RES0**.

## D1.2.49 DSCSR, Debug Security Control and Status Register

The DSCSR characteristics are:

### Purpose

Provides control and status information for Secure debug.

### Usage constraints

Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are **UNPREDICTABLE**.

This register is accessible only to the debugger and **RES0** to software.

### Configurations

Present only if the Security Extension is implemented.

This register is **RES0** if the Security Extension is not implemented.

Present only if Halting debug is implemented.

This register is **RES0** if Halting debug is not implemented.

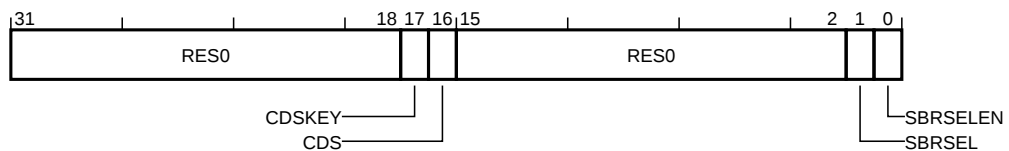
### Attributes

32-bit read/write register located at 0xE000EE08.

This register is not banked between Security states.

## Field descriptions

The DSCSR bit assignments are:



### Bits [31:18]

Reserved, **RES0**.

### CDSKEY, bit [17]

CDS write-enable key. Writes to the CDS bit are ignored unless CDSKEY is concurrently written to zero.

The possible values of this bit are:

- 0**  
Concurrent write to CDS not ignored.
- 1**  
Concurrent write to CDS ignored.

This bit reads-as-one.

### CDS, bit [16]

Current domain Secure. This field indicates the current Security state of the processor.

The possible values of this bit are:

- 0**  
PE is in Non-secure state.
- 1**  
PE is in Secure state.

This bit is only writable if DHCSR.S\_SDE is 1, the access to the register originates from the debugger, the PE is halted in Debug state, and CDSKEY is concurrently written to zero.

**Bits [15:2]**

Reserved, RES0.

**SBRSEL, bit [1]**

Secure banked register select. If SBRSELEN is 1 this bit selects whether the Non-secure or the Secure versions of the memory-mapped banked registers are accessible to the debugger.

The possible values of this bit are:

**0**  
Selects the Non-secure versions.

**1**  
Selects the Secure versions.

This bit behaves as RAZ/WI if DHCSR.S\_SDE is 0.

This bit resets to zero on a Cold reset.

**SBRSELEN, bit [0]**

Secure banked register select enable. Controls whether the SBRSEL field or the current Security state of the processor selects which version of the memory-mapped banked registers are accessible to the debugger.

The possible values of this bit are:

**0**  
The current Security state of the PE determines which memory-mapped Banked registers are accessed by the debugger.

**1**  
DSCSR.SBRSEL selects which memory-mapped Banked registers are accessed by the debugger.

This bit behaves as RAO/WI if DHCSR.S\_SDE is 0.

This bit resets to zero on a Cold reset.

**Note**

This method of banked register selection means that the register aliasing is not used for accesses from the debugger. Accesses to the aliased addresses from the debugger have the same behavior as reserved addresses.

## D1.2.50 DWT\_CIDR0, DWT Component Identification Register 0

The DWT\_CIDR0 characteristics are:

### Purpose

Provides CoreSight Unique Component Identifier information for the DWT.

### Usage constraints

Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are **UNPREDICTABLE**.

If the Main Extension is not implemented then it is **IMPLEMENTATION DEFINED** whether this register is accessible only to the debugger and **RES0** for software. Otherwise the register is accessible to the debugger and software.

### Configurations

Present only if CoreSight identification is implemented.

This register is **RES0** if CoreSight identification is not implemented.

Present only if the DWT is implemented.

This register is **RES0** if the DWT is not implemented.

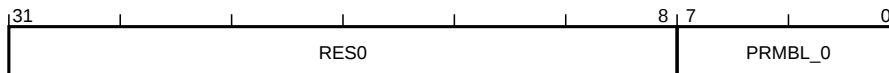
### Attributes

32-bit read-only register located at 0xE0001FF0.

This register is not banked between Security states.

## Field descriptions

The DWT\_CIDR0 bit assignments are:



### Bits [31:8]

Reserved, **RES0**.

### PRMBL\_0, bits [7:0]

CoreSight component identification preamble. See the *Arm® CoreSight™ Architecture Specification*.

This field reads as 0x0D.

## D1.2.51 DWT\_CIDR1, DWT Component Identification Register 1

The DWT\_CIDR1 characteristics are:

### Purpose

Provides CoreSight Unique Component Identifier information for the DWT.

### Usage constraints

Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are **UNPREDICTABLE**.

If the Main Extension is not implemented then it is **IMPLEMENTATION DEFINED** whether this register is accessible only to the debugger and **RES0** for software. Otherwise the register is accessible to the debugger and software.

### Configurations

Present only if CoreSight identification is implemented.

This register is **RES0** if CoreSight identification is not implemented.

Present only if the DWT is implemented.

This register is **RES0** if the DWT is not implemented.

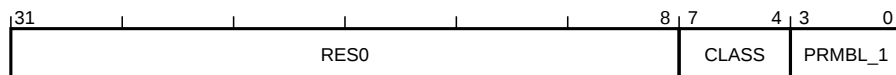
### Attributes

32-bit read-only register located at 0xE0001FF4.

This register is not banked between Security states.

## Field descriptions

The DWT\_CIDR1 bit assignments are:



### Bits [31:8]

Reserved, **RES0**.

### CLASS, bits [7:4]

CoreSight component class. See the *Arm® CoreSight™ Architecture Specification*.

This field reads as 0x9.

### PRMBL\_1, bits [3:0]

CoreSight component identification preamble. See the *Arm® CoreSight™ Architecture Specification*.

This field reads as 0x0.

## D1.2.52 DWT\_CIDR2, DWT Component Identification Register 2

The DWT\_CIDR2 characteristics are:

### Purpose

Provides CoreSight Unique Component Identifier information for the DWT.

### Usage constraints

Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are **UNPREDICTABLE**.

If the Main Extension is not implemented then it is **IMPLEMENTATION DEFINED** whether this register is accessible only to the debugger and **RES0** for software. Otherwise the register is accessible to the debugger and software.

### Configurations

Present only if CoreSight identification is implemented.

This register is **RES0** if CoreSight identification is not implemented.

Present only if the DWT is implemented.

This register is **RES0** if the DWT is not implemented.

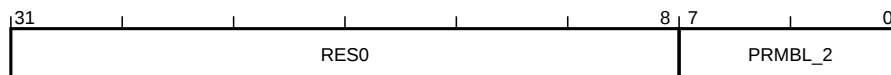
### Attributes

32-bit read-only register located at 0xE0001FF8.

This register is not banked between Security states.

## Field descriptions

The DWT\_CIDR2 bit assignments are:



### Bits [31:8]

Reserved, **RES0**.

### PRMBL\_2, bits [7:0]

CoreSight component identification preamble. See the *Arm® CoreSight™ Architecture Specification*.

This field reads as 0x05.



### D1.2.53 DWT\_CIDR3, DWT Component Identification Register 3

The DWT\_CIDR3 characteristics are:

#### Purpose

Provides CoreSight Unique Component Identifier information for the DWT.

#### Usage constraints

Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are **UNPREDICTABLE**.

If the Main Extension is not implemented then it is **IMPLEMENTATION DEFINED** whether this register is accessible only to the debugger and **RES0** for software. Otherwise the register is accessible to the debugger and software.

#### Configurations

Present only if CoreSight identification is implemented.

This register is **RES0** if CoreSight identification is not implemented.

Present only if the DWT is implemented.

This register is **RES0** if the DWT is not implemented.

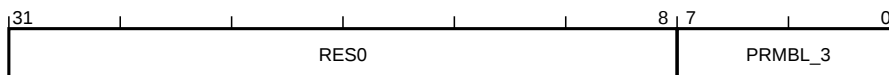
#### Attributes

32-bit read-only register located at 0xE0001FFC.

This register is not banked between Security states.

### Field descriptions

The DWT\_CIDR3 bit assignments are:



#### Bits [31:8]

Reserved, **RES0**.

#### PRMBL\_3, bits [7:0]

CoreSight component identification preamble. See the *Arm® CoreSight™ Architecture Specification*.

This field reads as 0xB1.

### D1.2.54 DWT\_COMPn, DWT Comparator Register, n = 0 - 14

The DWT\_COMP{0..14} characteristics are:

**Purpose**

Provides a reference value for use by watchpoint comparator *n*.

**Usage constraints**

Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

If the Main Extension is not implemented then it is IMPLEMENTATION DEFINED whether this register is accessible only to the debugger and RES0 for software. Otherwise the register is accessible to the debugger and software.

**Configurations**

Present only if the DWT is implemented.

This register is RES0 if the DWT is not implemented.

**Attributes**

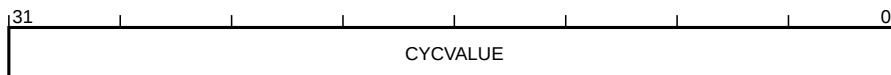
32-bit read/write register located at 0xE0001020 + 16*n*.

This register is not banked between Security states.

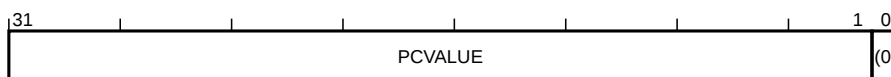
### Field descriptions

The DWT\_COMP{0..14} bit assignments are:

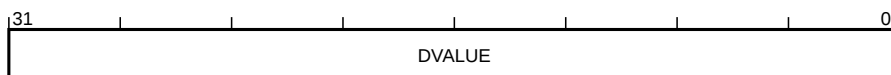
**When DWT\_FUNCTIONn.MATCH == 0b0001:**



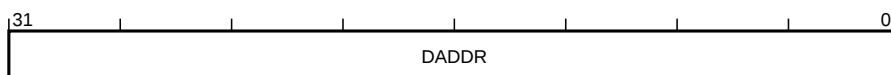
**When DWT\_FUNCTIONn.MATCH == 0b001x:**



**When DWT\_FUNCTIONn.MATCH == 0b10xx:**



**When DWT\_FUNCTIONn.MATCH == 0bx1xx:**



**CYCVALUE, bits [31:0], when DWT\_FUNCTIONn.MATCH == 0b0001**

Cycle value. Reference value for comparison with cycle count.

This field resets to an UNKNOWN value on a Cold reset.

**PCVALUE, bits [31:1], when DWT\_FUNCTIONn.MATCH == 0b001x**

PC value. Reference value for comparison with Program Counter.

This field resets to an **UNKNOWN** value on a Cold reset.

**Bit [0], when DWT\_FUNCTIONn.MATCH == 0b001x**

Reserved, **RES0**.

**DADDR, bits [31:0], when DWT\_FUNCTIONn.MATCH == 0bx1xx**

Data address. Reference value for comparison with load or store address.

For halfword address comparisons, DADDR[0] is **RES0**. For byte address comparisons, DADDR[1:0] are **RES0**.

This field resets to an **UNKNOWN** value on a Cold reset.

**DVALUE, bits [31:0], when DWT\_FUNCTIONn.MATCH == 0b10xx**

Data value. Reference value for comparison with load or store data.

For halfword or word comparisons, the data value is in little-endian order. That is, the least significant byte of this register is compared with the byte targeting the lowest address in memory.

For byte or halfword comparisons, if the value of the byte or halfword is not replicated across all byte or halfword lanes, the value used for the comparison is **UNKNOWN**.

This field resets to an **UNKNOWN** value on a Cold reset.

## D1.2.55 DWT\_CPICNT, DWT CPI Count Register

The DWT\_CPICNT characteristics are:

### Purpose

Counts additional cycles required to execute multicycle instructions and instruction fetch stalls.

### Usage constraints

Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are [UNPREDICTABLE](#).

### Configurations

Present only if the Main Extension is implemented.

This register is [RES0](#) if the Main Extension is not implemented.

Present only if the DWT is implemented.

This register is [RES0](#) if the DWT is not implemented.

Present only if `DWT_CTRL.NOPRFCNT == 0`.

This register is [RES0](#) if `DWT_CTRL.NOPRFCNT == 1`.

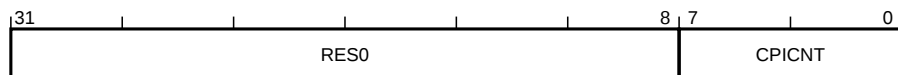
### Attributes

32-bit read/write register located at `0xE0001008`.

This register is not banked between Security states.

## Field descriptions

The DWT\_CPICNT bit assignments are:



### Bits [31:8]

Reserved, [RES0](#).

### CPICNT, bits [7:0]

Base instruction overhead counter.

Counts one on each cycle when all of the following are true:

- `DWT_CTRL.CPIEVTENA == 1` and `DEMCR.TRCENA == 1`.
- No instruction is executed.
- No load-store operation is in progress, see `DWT_LSUCNT`.
- No exception-entry or exception-exit operation is in progress, see `DWT_EXCCNT`.
- The PE is not in a power-saving mode, see `DWT_SLEEPCNT`.
- Either `SecureNoninvasiveDebugAllowed() == TRUE`, or the PE is in Non-secure state and `NoninvasiveDebugAllowed() == TRUE`.

The definition of "no instruction is executed" is [IMPLEMENTATION DEFINED](#). Arm recommends that this counts each cycle on which no instruction is retired.

Initialized to zero when the counter is disabled and `DWT_CTRL.CPIEVTENA` is written with 1. An Event Counter packet is emitted on counter overflow.

This field resets to an [UNKNOWN](#) value on a Cold reset.

## D1.2.56 DWT\_CTRL, DWT Control Register

The DWT\_CTRL characteristics are:

### Purpose

Provides configuration and status information for the DWT unit, and used to control features of the unit.

### Usage constraints

Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are **UNPREDICTABLE**.

If the Main Extension is not implemented then it is **IMPLEMENTATION DEFINED** whether this register is accessible only to the debugger and **RES0** for software. Otherwise the register is accessible to the debugger and software.

### Configurations

Present only if the DWT is implemented.

This register is **RES0** if the DWT is not implemented.

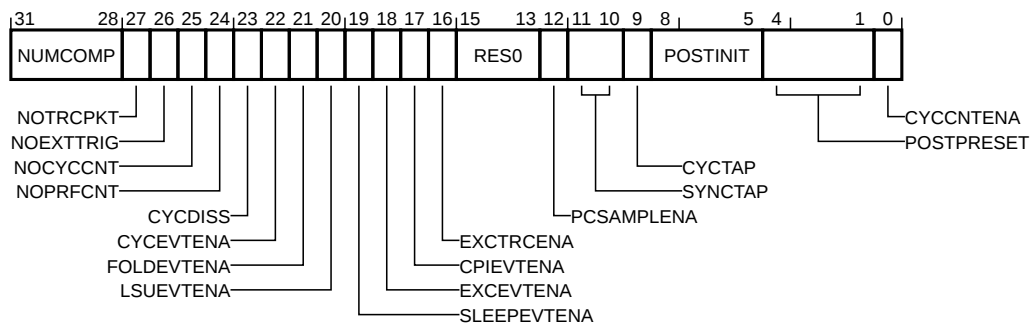
### Attributes

32-bit read/write register located at 0xE0001000.

This register is not banked between Security states.

## Field descriptions

The DWT\_CTRL bit assignments are:



### NUMCOMP, bits [31:28]

Number of comparators. Number of DWT comparators implemented.

A value of zero indicates no comparator support.

This field reads as an **IMPLEMENTATION DEFINED** value.

### NOTRCPKT, bit [27]

No trace packets. Indicates whether the implementation does not support trace.

The possible values of this bit are:

**0**  
Trace supported.

**1**  
Trace not supported.

If this bit is RAZ, the NOCYCCNT bit must also be RAZ.

If the Main Extension is not implemented, this bit is **RES1**.

This bit reads as an **IMPLEMENTATION DEFINED** value.

**NOEXTTRIG, bit [26]**

No External Triggers. Shows whether the implementation does not support external triggers.

Reserved, **RES0**.

**NOCYCCNT, bit [25]**

No cycle count. Indicates whether the implementation does not include a cycle counter.

The possible values of this bit are:

**0**

Cycle counter implemented.

**1**

Cycle counter not implemented.

If the Main Extension is not implemented, this bit is **RES1**.

This bit reads as an **IMPLEMENTATION DEFINED** value.

**NOPRFCNT, bit [24]**

No profile counters. Indicates whether the implementation does not include the profiling counters.

The possible values of this bit are:

**0**

Profiling counters implemented.

**1**

Profiling counters not implemented.

If the Main Extension is not implemented, this bit is **RES1**.

This bit reads as an **IMPLEMENTATION DEFINED** value.

**CYCDISS, bit [23]**

Cycle counter disabled secure. Controls whether the cycle counter is disabled in Secure state.

The possible values of this bit are:

**0**

No effect.

**1**

Disable incrementing of the cycle counter when the PE is in Secure state.

**RES0** if the NOCYCCNT bit is RAO.

If the Main Extension is not implemented, this bit is **RES0**.

This bit resets to zero on a Cold reset.

**CYCEVTENA, bit [22]**

Cycle event enable. Enables Event Counter packet generation on POSTCNT underflow.

The possible values of this bit are:

**0**

No Event Counter packets generated when POSTCNT underflows.

**1**

If PCSAMPLENA set to 0, an Event Counter packet is generated when POSTCNT underflows.

**RES0** if the NOTRCPKT bit is RAO or the NOCYCCNT bit is RAO.

If the Main Extension is not implemented, this bit is **RES0**.

This bit resets to zero on a Cold reset.

**FOLDEVTENA, bit [21]**

Fold event enable. Enables DWT\_FOLDCNT counter.

The possible values of this bit are:

**0**  
DWT\_FOLDCNT disabled.

**1**  
DWT\_FOLDCNT enabled.

**RES0** if the NOPRFCNT bit is RAO. The reset value is 0.

If the Main Extension is not implemented, this bit is **RES0**.

This bit resets to zero on a Cold reset.

**LSUEVTENA, bit [20]**

LSU event enable. Enables DWT\_LSUCNT counter.

The possible values of this bit are:

**0**  
DWT\_LSUCNT disabled.

**1**  
DWT\_LSUCNT enabled.

**RES0** if the NOPRFCNT bit is RAO.

If the Main Extension is not implemented, this bit is **RES0**.

This bit resets to zero on a Cold reset.

**SLEEPEVTENA, bit [19]**

Sleep event enable. Enable DWT\_SLEEPCNT counter.

The possible values of this bit are:

**0**  
DWT\_SLEEPCNT disabled.

**1**  
DWT\_SLEEPCNT enabled.

**RES0** if the NOPRFCNT bit is RAO.

If the Main Extension is not implemented, this bit is **RES0**.

This bit resets to zero on a Cold reset.

**EXCEVTENA, bit [18]**

Exception event enable. Enables DWT\_EXCCNT counter.

The possible values of this bit are:

**0**  
DWT\_EXCCNT disabled.

**1**  
DWT\_EXCCNT enabled.

**RES0** if the NOPRFCNT bit is RAO.

If the Main Extension is not implemented, this bit is **RES0**.

This bit resets to zero on a Cold reset.

**CPIEVTENA, bit [17]**

CPI event enable. Enables DWT\_CPICNT counter.

The possible values of this bit are:

**0**  
DWT\_CPICNT disabled.

**1**  
DWT\_CPICNT enabled.

**RES0** if the NOPRFCNT bit is RAO.

If the Main Extension is not implemented, this bit is **RES0**.

This bit resets to zero on a Cold reset.

**EXCTRCENA, bit [16]**

Exception trace enable. Enables generation of Exception Trace packets.

The possible values of this bit are:

**0**  
Exception Trace packet generation disabled.

**1**  
Exception Trace packet generation enabled.

**RES0** if the NOTRCPKT bit is RAO.

If the Main Extension is not implemented, this bit is **RES0**.

This bit resets to zero on a Cold reset.

**Bits [15:13]**

Reserved, **RES0**.

**PCSAMPLENA, bit [12]**

PC sample enable. Enables use of POSTCNT counter as a timer for Periodic PC Sample packet generation.

The possible values of this bit are:

**0**  
Periodic PC Sample packet generation disabled.

**1**  
Periodic PC Sample packet generated on POSTCNT underflow.

**RES0** if the NOTRCPKT bit is RAO or the NOCYCCNT bit is RAO.

If the Main Extension is not implemented, this bit is **RES0**.

This bit resets to zero on a Cold reset.

**SYNCTAP, bits [11:10]**

Synchronization tap. Selects the position of the synchronization packet request counter tap on the CYCCNT counter. This determines the rate of Synchronization packet requests made by the DWT.

The possible values of this field are:

**0b00**  
Synchronization packet request disabled.

**0b01**  
Synchronization counter tap at CYCCNT[24].

**0b10**  
Synchronization counter tap at CYCCNT[26].



**0b11**

Synchronization counter tap at CYCCNT[28].

**RES0** if the NOCYCCNT bit is RAO.

If the Main Extension is not implemented, this field is **RES0**.

This field resets to an **UNKNOWN** value on a Cold reset.

**CYCTAP, bit [9]**

Cycle count tap. Selects the position of the POSTCNT tap on the CYCCNT counter.

The possible values of this bit are:

**0**

POSTCNT tap at CYCCNT[6].

**1**

POSTCNT tap at CYCCNT[10].

**RES0** if the NOCYCCNT bit is RAO.

If the Main Extension is not implemented, this bit is **RES0**.

This bit resets to an **UNKNOWN** value on a Cold reset.

**POSTINIT, bits [8:5]**

POSTCNT initial. Initial value for the POSTCNT counter.

**RES0** if the NOCYCCNT bit is RAO.

If the Main Extension is not implemented, this field is **RES0**.

This field resets to an **UNKNOWN** value on a Cold reset.

**POSTPRESET, bits [4:1]**

POSTCNT preset. Reload value for the POSTCNT counter.

**RES0** if the NOCYCCNT bit is RAO.

If the Main Extension is not implemented, this field is **RES0**.

This field resets to an **UNKNOWN** value on a Cold reset.

**CYCCNTENA, bit [0]**

CYCCNT enable. Enables CYCCNT.

The possible values of this bit are:

**0**

CYCCNT disabled.

**1**

CYCCNT enabled.

**RES0** if the NOCYCCNT bit is RAO.

If the Main Extension is not implemented, this bit is **RES0**.

This bit resets to zero on a Cold reset.

## D1.2.57 DWT\_CYCCNT, DWT Cycle Count Register

The DWT\_CYCCNT characteristics are:

### Purpose

Shows or sets the value of the processor cycle counter, CYCCNT.

### Usage constraints

Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are [UNPREDICTABLE](#).

### Configurations

Present only if the Main Extension is implemented.

This register is [RES0](#) if the Main Extension is not implemented.

Present only if the DWT is implemented.

This register is [RES0](#) if the DWT is not implemented.

Present only if `DWT_CTRL.NOCYCCNT == 0`.

This register is [RES0](#) if `DWT_CTRL.NOCYCCNT == 1`.

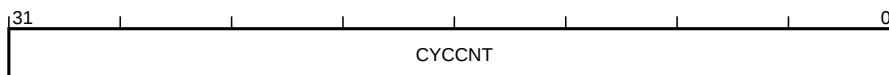
### Attributes

32-bit read/write register located at `0xE0001004`.

This register is not banked between Security states.

## Field descriptions

The DWT\_CYCCNT bit assignments are:



### CYCCNT, bits [31:0]

Incrementing cycle counter value. Increments one on each processor clock cycle when `DWT_CTRL.CYCCNTENA == 1` and `DEMCR.TRCENA == 1`. On overflow, CYCCNT wraps to zero.

This field resets to an [UNKNOWN](#) value on a Cold reset.

## D1.2.58 DWT\_DEVARCH, DWT Device Architecture Register

The DWT\_DEVARCH characteristics are:

### Purpose

Provides CoreSight Unique Component Identifier information for the DWT.

### Usage constraints

Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are **UNPREDICTABLE**.

If the Main Extension is not implemented then it is **IMPLEMENTATION DEFINED** whether this register is accessible only to the debugger and **RES0** for software. Otherwise the register is accessible to the debugger and software.

### Configurations

Present only if CoreSight identification is implemented.

This register is **RES0** if CoreSight identification is not implemented.

Present only if the DWT is implemented.

This register is **RES0** if the DWT is not implemented.

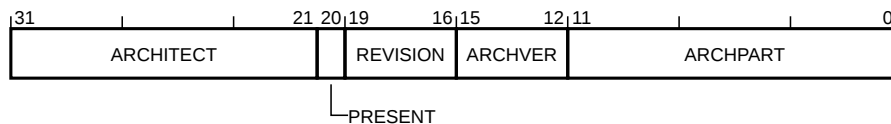
### Attributes

32-bit read-only register located at 0xE0001FBC.

This register is not banked between Security states.

## Field descriptions

The DWT\_DEVARCH bit assignments are:



### ARCHITECT, bits [31:21]

Architect. Defines the architect of the component. Bits [31:28] are the JEP106 continuation code (JEP106 bank ID, minus 1) and bits [27:21] are the JEP106 ID code.

The possible values of this field are:

#### 0x23B

JEP106 continuation code 0x4, ID code 0x3B. Arm Limited.

Other values are defined by the JEDEC JEP106 standard.

This field reads as 0x23B.

### PRESENT, bit [20]

DEVARCH Present. Defines that the DEVARCH register is present.

The possible values of this bit are:

#### 1

DEVARCH information present.

This bit reads as one.

**REVISION, bits [19:16]**

Revision. Defines the architecture revision of the component.

The possible values of this field are:

**0b0000**

DWT architecture v2.0.

This field reads as 0b0000.

**ARCHVER, bits [15:12]**

Architecture Version. Defines the architecture version of the component.

The possible values of this field are:

**0b0001**

DWT architecture v2.

ARCHVER and ARCHPART are also defined as a single field, ARCHID, so that ARCHVER is ARCHID[15:12].

This field reads as 0b0001.

**ARCHPART, bits [11:0]**

Architecture Part. Defines the architecture of the component.

The possible values of this field are:

**0xA02**

DWT architecture.

ARCHVER and ARCHPART are also defined as a single field, ARCHID, so that ARCHPART is ARCHID[11:0].

This field reads as 0xA02.



## D1.2.60 DWT\_EXCCNT, DWT Exception Overhead Count Register

The DWT\_EXCCNT characteristics are:

### Purpose

Counts the total cycles spent in exception processing.

### Usage constraints

Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are **UNPREDICTABLE**.

### Configurations

Present only if the Main Extension is implemented.

This register is **RES0** if the Main Extension is not implemented.

Present only if the DWT is implemented.

This register is **RES0** if the DWT is not implemented.

Present only if `DWT_CTRL.NOPRFCNT == 0`.

This register is **RES0** if `DWT_CTRL.NOPRFCNT == 1`.

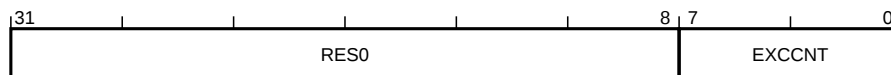
### Attributes

32-bit read/write register located at `0xE000100C`.

This register is not banked between Security states.

## Field descriptions

The DWT\_EXCCNT bit assignments are:



### Bits [31:8]

Reserved, **RES0**.

### EXCCNT, bits [7:0]

The exception overhead counter.

Counts one on each cycle when all of the following are true:

- `DWT_CTRL.EXCEVTENA == 1` and `DEMCR.TRCENA == 1`.
- No instruction is executed, see `DWT_CPICNT`.
- An exception-entry or exception-exit related operation is in progress.
- Either `SecureNoninvasiveDebugAllowed() == TRUE`, or `NS-Req` for the operation is set to Non-secure and `NoninvasiveDebugAllowed() == TRUE`.

Exception-entry or exception-exit related operations include the stacking of registers on exception entry, lazy state preservation, unstacking of registers on exception exit, and preemption.

Initialized to zero when the counter is disabled and `DWT_CTRL.EXCEVTENA` is written with 1. An Event Counter packet is emitted on counter overflow.

This field resets to an **UNKNOWN** value on a Cold reset.

## D1.2.61 DWT\_FOLDCNT, DWT Folded Instruction Count Register

The DWT\_FOLDCNT characteristics are:

### Purpose

Increments for each additional instruction executed in the current cycle.

### Usage constraints

Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are [UNPREDICTABLE](#).

### Configurations

Present only if the Main Extension is implemented.

This register is [RES0](#) if the Main Extension is not implemented.

Present only if the DWT is implemented.

This register is [RES0](#) if the DWT is not implemented.

Present only if `DWT_CTRL.NOPRFCNT == 0`.

This register is [RES0](#) if `DWT_CTRL.NOPRFCNT == 1`.

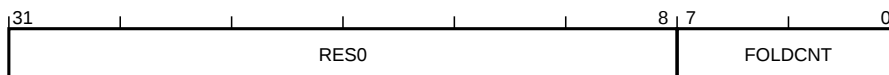
### Attributes

32-bit read/write register located at `0xE0001018`.

This register is not banked between Security states.

## Field descriptions

The DWT\_FOLDCNT bit assignments are:



### Bits [31:8]

Reserved, [RES0](#).

### FOLDCNT, bits [7:0]

Folded instruction counter.

Counts on each cycle when all of the following are true:

- `DWT_CTRL.FOLDEVTENA == 1` and `DEMCR.TRCENA == 1`.
- At least two instructions are executed, see `DWT_CPICNT`.
- Either `SecureNoninvasiveDebugAllowed() == TRUE`, or the PE is in Non-secure state and `NoninvasiveDebugAllowed() == TRUE`.

The counter is incremented by the number of instructions executed, minus one.

Initialized to zero when the counter is disabled and `DWT_CTRL.FOLDEVTENA` is written with 1. An event is emitted on counter overflow.

This field resets to an [UNKNOWN](#) value on a Cold reset.

### D1.2.62 DWT\_FUNCTIONn, DWT Comparator Function Register, n = 0 - 14

The DWT\_FUNCTION{0..14} characteristics are:

**Purpose**

Controls the operation of watchpoint comparator *n*.

**Usage constraints**

Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

If the Main Extension is not implemented then it is IMPLEMENTATION DEFINED whether this register is accessible only to the debugger and RES0 for software. Otherwise the register is accessible to the debugger and software.

**Configurations**

Present only if the DWT is implemented.

This register is RES0 if the DWT is not implemented.

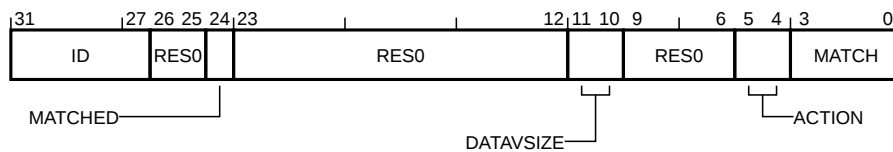
**Attributes**

32-bit read/write register located at 0xE0001028 + 16*n*.

This register is not banked between Security states.

### Field descriptions

The DWT\_FUNCTION{0..14} bit assignments are:



**ID, bits [31:27]**

Identify capability. Identifies the capabilities for MATCH for comparator *n*.

The possible values of this field are:

**0b00000**

Reserved.

**0b01000**

Data Address, and Data Address With Value.

**0b01001**

Cycle Counter, Data Address, and Data Address With Value.

**0b01010**

Instruction Address, Data Address, and Data Address With Value.

**0b01011**

Cycle Counter, Instruction Address, Data Address and Data Address With Value.

**0b11000**

Data Address, Data Address Limit, and Data Address With Value.

**0b11010**

Instruction Address, Instruction Address Limit, Data Address, Data Address Limit, and Data Address With Value.



**0b111100**

Data Address, Data Address Limit, Data Value, Linked Data Value, and Data Address With Value.

**0b111110**

Instruction Address, Instruction Address Limit, Data Address, Data Address Limit, Data value, Linked Data Value, and Data Address With Value.

All other values are reserved.

Comparator 0 never supports linking. If more than one comparator is implemented, then at least one comparator must support linking. Arm recommends that odd-numbered comparators support linking.

Cycle Counter matching is only supported if the Main Extension is implemented and `DWT_CTRL.NOCYCCNT == 0`, meaning the cycle counter is implemented. Comparator 0 must support Cycle Counter matching if the cycle counter is implemented.

Data Address With Value is supported for the first four comparators only, and only if the Main Extension and ITM are implemented, and `DWT_CTRL.NOTRCPKT == 0`. Data Value and Linked Data Value not supported if the Main Extension is not implemented.

This field is read-only.

This field reads as an **IMPLEMENTATION DEFINED** value.

**Bits [26:25]**

Reserved, **RES0**.

**MATCHED, bit [24]**

Comparator matched. Set to 1 when the comparator matches.

The possible values of this bit are:

**0**

No match.

**1**

Match. The comparator has matched since the last read of this register.

For an Instruction Address Limit or Data Address Limit comparator, this bit is **UNKNOWN** on reads.

This bit is read-only.

This bit clears to zero when read.

This bit resets to an **UNKNOWN** value on a Cold reset.

**Bits [23:12]**

Reserved, **RES0**.

**DATAVSIZ, bits [11:10]**

Data value size. Defines the size of the object being watched for by Data Value and Data Address comparators.

The possible values of this field are:

**0b00**

1 byte.

**0b01**

2 bytes.

**0b10**

4 bytes.

All other values are reserved.

For an Instruction Address or Instruction Address Limit comparator, **DATAVSIZ** must be `0b01` (2 bytes). If this comparator is part of an data address range pair, **DATAVSIZ** must be `0b00` (1 byte).

For a Data Address comparator, DWT\_COMP $n$  must be aligned to the size specified by DATAVSIZE. For a Data Value or Linked Data Value comparator:

- For halfword comparisons, DWT\_COMP $n$  [31:16] must be equal to DWT\_COMP $n$ [15:0].
- For byte comparisons, DWT\_COMP $n$  [31:24], DWT\_COMP $n$  [23:16], and DWT\_COMP $n$  [15:18] must be equal to DWT\_COMP $n$  [7:0].

This field resets to an UNKNOWN value on a Cold reset.

**Bits [9:6]**

Reserved, RES0.

**ACTION, bits [5:4]**

Action on match. Defines the action on a match. This field is ignored and the comparator generates no actions if it is disabled by MATCH.

The possible values of this field are:

**0b00**

Trigger only.

**0b01**

Generate debug event.

**0b10**

For a Cycle Counter, Instruction Address, Data Address, Data Value or Linked Data Value comparator, generate a Data Trace Match packet.

For a Data Address With Value comparator, generate a Data Trace Data Value packet.

**0b11**

For a Data Address Limit comparator, generate a Data Trace Data Address packet.

For a Cycle Counter, Instruction Address Limit, or Data Address comparator, generate a Data Trace PC Value packet.

For a Data Address With Value comparator, generate both a Data Trace PC Value packet and a Data Trace Data Value packet.

If the Main Extension is not implemented, the values 0b10 and 0b11 are reserved.

This field resets to an UNKNOWN value on a Cold reset.

**MATCH, bits [3:0]**

Match type. Controls the type of match generated by this comparator.

The possible values of this field are:

**0b0000**

Disabled. Never generates a match.

**0b0001**

Cycle Counter. Matches if DWT\_CYCCNT equals the comparator value. The comparator is checked each time DWT\_CYCCNT is written to, directly or indirectly.

Only supported if the Main Extension is implemented, DWT\_FUNCTION< $n$ >.ID<0> == 1 and DWT\_CTRL.NOCYCCNT == 0.

**0b0010**

Instruction Address. If not linked to, an instruction matches if the address of the first byte of the instruction matches the comparator address.

Only supported if DWT\_FUNCTION< $n$ >.ID<1> == 1.

**0b0011**

Instruction Address Limit. An instruction matches if the address of the first byte of the instruction lies between the lower comparator address (specified by comparator  $\langle n-1 \rangle$ ) and the limit comparator address (specified by this comparator,  $\langle n \rangle$ ). Both addresses are inclusive to the range. Comparator  $\langle n-1 \rangle$  must be programmed for Instruction Address (0b0010) or Disabled (0b0000), and the lower address must be strictly less-than the limit comparator address, otherwise it is **UNPREDICTABLE** whether or not any comparator generates matches.

Only supported if  $DWT\_FUNCTION\langle n \rangle.ID\langle 4 \rangle == 1$  and  $DWT\_FUNCTION\langle n \rangle.ID\langle 1 \rangle == 1$ .

**0b0100**

Data Address. If not linked to by a Data Address Limit comparator, an access matches if any accessed byte lies between the comparator address and a limit defined by the DATAVSIZ field. Supported for all comparators.

**0b0101**

Data Address, writes. As 0b0100, except that only write accesses generate a match.

**0b0110**

Data Address, reads. As 0b0100, except that only read accesses generate a match.

**0b0111**

Data Address Limit. An access matches if any byte made by the access lies between the lower address (specified by comparator  $\langle n-1 \rangle$ ) and the limit address (specified by this comparator,  $\langle n \rangle$ ). Both addresses are inclusive to the range. Comparator  $\langle n-1 \rangle$  must be programmed for Data Address (0b01xx, not 0b0111), Data Address With Data Value (0b11xx, not 0b1111), or Disabled (0b0000), and the lower address must be strictly less-than the limit comparator address, otherwise it is **UNPREDICTABLE** whether or not any comparator generates matches.  $DWT\_FUNCTION\langle n-1 \rangle.MATCH[1:0]$  determines the matching access types.

Only supported if  $DWT\_FUNCTION\langle n \rangle.ID\langle 4 \rangle == 1$ .

**0b1000**

Data Value. An access matches if the value accessed matches the comparator value.

Only supported if the Main Extension is implemented and  $DWT\_FUNCTION\langle n \rangle.ID\langle 2 \rangle == 1$ .

**0b1001**

Data Value, writes. As 0b1000, except that only write accesses generate a match.

**0b1010**

Data Value, reads. As 0b1000, except that only read accesses generate a match.

**0b1011**

Linked Data Value. An access matches if the value accessed matches the comparator value (specified by comparator  $\langle n \rangle$ ) and the linked data address (specified by comparator  $\langle n-1 \rangle$ ) for the same access matches. Comparator  $\langle n-1 \rangle$  must be programmed for Data Address (0b01xx, not 0b0111), or Data Address With Value (0b11xx, not 0b1111), or Disabled (0b0000), and DATAVSIZ for the two comparators must be the same, otherwise it is **UNPREDICTABLE** whether or not any comparator generates matches.  $DWT\_FUNCTION\langle n-1 \rangle.MATCH[1:0]$  determines the matching access types.

Only supported if the Main Extension is implemented and  $DWT\_FUNCTION\langle n \rangle.ID\langle 4 \rangle == 1$  and  $DWT\_FUNCTION\langle n \rangle.ID\langle 2 \rangle == 1$ .

**0b1100**

Data Address With Value. As 0b01xx, except that the data value is traced.

Supported for the first four comparators only, and only if  $DWT\_CTRL.NOTRCPKT == 0$  and ITM is also implemented.

**0b1101**

Data Address With Value, writes. As 0b1100, except that only write accesses generate a match.

**0b1110**

Data Address With Value, reads. As 0b1100, except that only read accesses generate a match.

Any value not supported by the comparator is reserved. For a pair of linked comparators,  $\langle n \rangle$  and  $\langle n-1 \rangle$ , DWT\_FUNCTION $\langle n-1 \rangle$ .MATCH[1:0] determines the matching access types. See MATCH table for further details.

This field resets to zero on a Cold reset.

### D1.2.63 DWT\_LAR, DWT Software Lock Access Register

The DWT\_LAR characteristics are:

#### Purpose

Provides CoreSight Software Lock control for the DWT, see the *Arm® CoreSight™ Architecture Specification* for details.

#### Usage constraints

Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are **UNPREDICTABLE**.

This register is RAZ/WI if accessed via the debugger.

#### Configurations

Present only if CoreSight identification is implemented.

This register is **RES0** if CoreSight identification is not implemented.

Present only if the DWT is implemented.

This register is **RES0** if the DWT is not implemented.

Present only if the optional Software Lock is implemented.

This register is RAZ/WI if the Software Lock is not implemented.

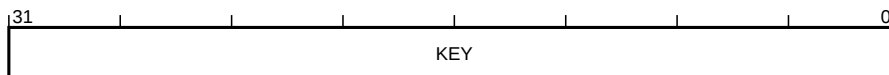
#### Attributes

32-bit write-only register located at 0xE0001FB0.

This register is not banked between Security states.

#### Field descriptions

The DWT\_LAR bit assignments are:



#### KEY, bits [31:0]

Lock Access control.

Writing the key value 0xC5ACCE55 to this field unlocks the lock, enabling write accesses to the registers of this component through a memory-mapped interface.

Writing any other value to this register locks the lock, disabling write accesses to the registers of this component through a memory mapped interface.

### D1.2.64 DWT\_LSR, DWT Software Lock Status Register

The DWT\_LSR characteristics are:

#### Purpose

Provides CoreSight Software Lock status information for the DWT, see the *Arm® CoreSight™ Architecture Specification* for details.

#### Usage constraints

Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are **UNPREDICTABLE**.

This register is RAZ/WI if accessed via the debugger.

#### Configurations

Present only if CoreSight identification is implemented.

This register is **RES0** if CoreSight identification is not implemented.

Present only if the DWT is implemented.

This register is **RES0** if the DWT is not implemented.

Present only if the optional Software Lock is implemented.

This register is RAZ/WI if the Software Lock is not implemented.

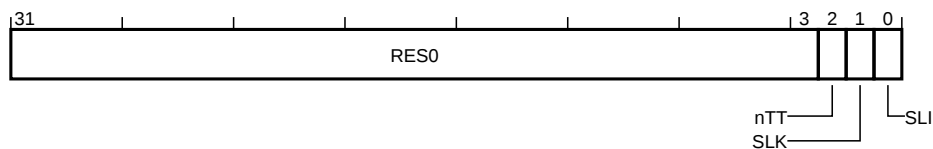
#### Attributes

32-bit read-only register located at 0xE0001FB4.

This register is not banked between Security states.

### Field descriptions

The DWT\_LSR bit assignments are:



#### Bits [31:3]

Reserved, **RES0**.

#### nTT, bit [2]

Not thirty-two bit. See the *Arm® CoreSight™ Architecture Specification*.

This bit reads as zero.

#### SLK, bit [1]

Software Lock status. See the *Arm® CoreSight™ Architecture Specification*.

The possible values of this bit are:

**0**

Lock clear. Software writes are permitted to the registers of this component.

**1**

Lock set. Software writes to the registers of this component are ignored, and reads have no side effects.

For a debugger read of this register, or when the Software Lock is not implemented, this bit is **RES0**.

This bit resets to one on a Cold reset.

**SLI, bit [0]**

Software Lock implemented. See the *Arm® CoreSight™ Architecture Specification*.

The possible values of this bit are:

**0**

Software Lock not implemented or debugger access.

**1**

Software Lock is implemented and software access.

For a debugger read of this register, or when the Software Lock is not implemented, this bit is **RAZ**.

This bit reads as an **IMPLEMENTATION DEFINED** value.





## D1.2.66 DWT\_PCSR, DWT Program Counter Sample Register

The DWT\_PCSR characteristics are:

### Purpose

Samples the current value of the Program Counter.

### Usage constraints

Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are **UNPREDICTABLE**.

If the Main Extension is not implemented then it is **IMPLEMENTATION DEFINED** whether this register is accessible only to the debugger and **RES0** for software. Otherwise the register is accessible to the debugger and software.

### Configurations

Present only if the DWT is implemented.

This register is **RES0** if the DWT is not implemented.

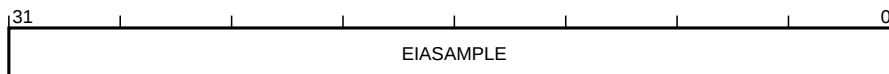
### Attributes

32-bit read-only register located at 0xE000101C.

This register is not banked between Security states.

## Field descriptions

The DWT\_PCSR bit assignments are:



### EIASAMPLE, bits [31:0]

Executed instruction address sample. Recently executed instruction address sample value.

The possible values of this field are:

#### 0xFFFFFFFF

One of the following is true:

- The PE is halted in Debug state.
- The Security Extension is implemented, the sampled instruction was executed in Secure state, and SecureNoninvasiveDebugAllowed() == FALSE.
- NoninvasiveDebugAllowed() == FALSE.
- DEMCR.TRCENA == 0.
- The address of a recently-executed instruction is not available.

#### Not 0xFFFFFFFF

Instruction address of a recently executed instruction. Bit [0] of the sample instruction address is 0.

The conditions when the address of a recently-executed instruction is not available are **IMPLEMENTATION DEFINED**.

## D1.2.67 DWT\_PIDR0, DWT Peripheral Identification Register 0

The DWT\_PIDR0 characteristics are:

### Purpose

Provides CoreSight Unique Component Identifier information for the DWT.

### Usage constraints

Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are **UNPREDICTABLE**.

If the Main Extension is not implemented then it is **IMPLEMENTATION DEFINED** whether this register is accessible only to the debugger and **RES0** for software. Otherwise the register is accessible to the debugger and software.

### Configurations

Present only if CoreSight identification is implemented.

This register is **RES0** if CoreSight identification is not implemented.

Present only if the DWT is implemented.

This register is **RES0** if the DWT is not implemented.

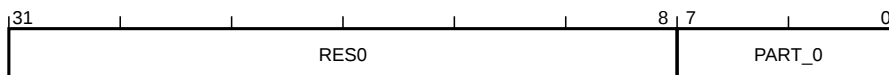
### Attributes

32-bit read-only register located at 0xE0001FE0.

This register is not banked between Security states.

## Field descriptions

The DWT\_PIDR0 bit assignments are:



### Bits [31:8]

Reserved, **RES0**.

### PART\_0, bits [7:0]

Part number bits [7:0]. See the *Arm® CoreSight™ Architecture Specification*.

This field reads as an **IMPLEMENTATION DEFINED** value.

## D1.2.68 DWT\_PIDR1, DWT Peripheral Identification Register 1

The DWT\_PIDR1 characteristics are:

### Purpose

Provides CoreSight Unique Component Identifier information for the DWT.

### Usage constraints

Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are **UNPREDICTABLE**.

If the Main Extension is not implemented then it is **IMPLEMENTATION DEFINED** whether this register is accessible only to the debugger and **RES0** for software. Otherwise the register is accessible to the debugger and software.

### Configurations

Present only if CoreSight identification is implemented.

This register is **RES0** if CoreSight identification is not implemented.

Present only if the DWT is implemented.

This register is **RES0** if the DWT is not implemented.

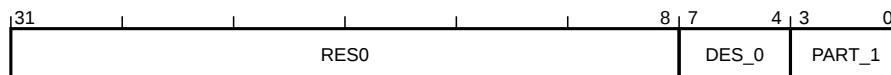
### Attributes

32-bit read-only register located at 0xE0001FE4.

This register is not banked between Security states.

## Field descriptions

The DWT\_PIDR1 bit assignments are:



### Bits [31:8]

Reserved, **RES0**.

### DES\_0, bits [7:4]

JEP106 identification code bits [3:0]. See the *Arm® CoreSight™ Architecture Specification*.

This field reads as an **IMPLEMENTATION DEFINED** value.

### PART\_1, bits [3:0]

Part number bits [11:8]. See the *Arm® CoreSight™ Architecture Specification*.

This field reads as an **IMPLEMENTATION DEFINED** value.

## D1.2.69 DWT\_PIDR2, DWT Peripheral Identification Register 2

The DWT\_PIDR2 characteristics are:

### Purpose

Provides CoreSight Unique Component Identifier information for the DWT.

### Usage constraints

Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are **UNPREDICTABLE**.

If the Main Extension is not implemented then it is **IMPLEMENTATION DEFINED** whether this register is accessible only to the debugger and **RES0** for software. Otherwise the register is accessible to the debugger and software.

### Configurations

Present only if CoreSight identification is implemented.

This register is **RES0** if CoreSight identification is not implemented.

Present only if the DWT is implemented.

This register is **RES0** if the DWT is not implemented.

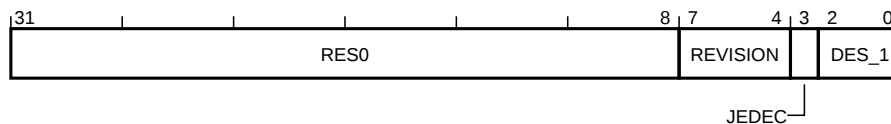
### Attributes

32-bit read-only register located at 0xE0001FE8.

This register is not banked between Security states.

## Field descriptions

The DWT\_PIDR2 bit assignments are:



### Bits [31:8]

Reserved, **RES0**.

### REVISION, bits [7:4]

Component revision. See the *Arm® CoreSight™ Architecture Specification*.

This field reads as an **IMPLEMENTATION DEFINED** value.

### JEDEC, bit [3]

JEDEC assignee value is used. See the *Arm® CoreSight™ Architecture Specification*.

This bit reads as one.

### DES\_1, bits [2:0]

JEP106 identification code bits [6:4]. See the *Arm® CoreSight™ Architecture Specification*.

This field reads as an **IMPLEMENTATION DEFINED** value.

### D1.2.70 DWT\_PIDR3, DWT Peripheral Identification Register 3

The DWT\_PIDR3 characteristics are:

#### Purpose

Provides CoreSight Unique Component Identifier information for the DWT.

#### Usage constraints

Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are **UNPREDICTABLE**.

If the Main Extension is not implemented then it is **IMPLEMENTATION DEFINED** whether this register is accessible only to the debugger and **RES0** for software. Otherwise the register is accessible to the debugger and software.

#### Configurations

Present only if CoreSight identification is implemented.

This register is **RES0** if CoreSight identification is not implemented.

Present only if the DWT is implemented.

This register is **RES0** if the DWT is not implemented.

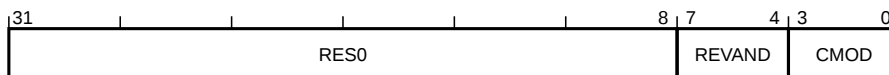
#### Attributes

32-bit read-only register located at 0xE0001FEC.

This register is not banked between Security states.

### Field descriptions

The DWT\_PIDR3 bit assignments are:



#### Bits [31:8]

Reserved, **RES0**.

#### REVAND, bits [7:4]

RevAnd. See the *Arm® CoreSight™ Architecture Specification*.

This field reads as an **IMPLEMENTATION DEFINED** value.

#### CMOD, bits [3:0]

Customer Modified. See the *Arm® CoreSight™ Architecture Specification*.

This field reads as an **IMPLEMENTATION DEFINED** value.

## D1.2.71 DWT\_PIDR4, DWT Peripheral Identification Register 4

The DWT\_PIDR4 characteristics are:

### Purpose

Provides CoreSight Unique Component Identifier information for the DWT.

### Usage constraints

Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are **UNPREDICTABLE**.

If the Main Extension is not implemented then it is **IMPLEMENTATION DEFINED** whether this register is accessible only to the debugger and **RES0** for software. Otherwise the register is accessible to the debugger and software.

### Configurations

Present only if CoreSight identification is implemented.

This register is **RES0** if CoreSight identification is not implemented.

Present only if the DWT is implemented.

This register is **RES0** if the DWT is not implemented.

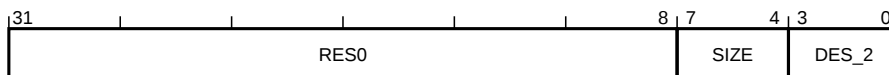
### Attributes

32-bit read-only register located at 0xE0001FD0.

This register is not banked between Security states.

## Field descriptions

The DWT\_PIDR4 bit assignments are:



### Bits [31:8]

Reserved, **RES0**.

### SIZE, bits [7:4]

4KB count. See the *Arm® CoreSight™ Architecture Specification*.

This field reads as zero.

### DES\_2, bits [3:0]

JEP106 continuation code. See the *Arm® CoreSight™ Architecture Specification*.

This field reads as an **IMPLEMENTATION DEFINED** value.

### D1.2.72 DWT\_PIDR5, DWT Peripheral Identification Register 5

The DWT\_PIDR5 characteristics are:

#### Purpose

Provides CoreSight Unique Component Identifier information for the DWT.

#### Usage constraints

Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

If the Main Extension is not implemented then it is IMPLEMENTATION DEFINED whether this register is accessible only to the debugger and RES0 for software. Otherwise the register is accessible to the debugger and software.

#### Configurations

Present only if CoreSight identification is implemented.

This register is RES0 if CoreSight identification is not implemented.

Present only if the DWT is implemented.

This register is RES0 if the DWT is not implemented.

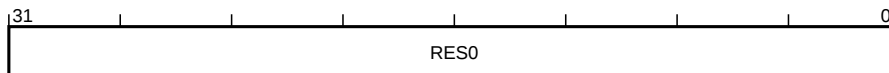
#### Attributes

32-bit read-only register located at 0xE0001FD4.

This register is not banked between Security states.

#### Field descriptions

The DWT\_PIDR5 bit assignments are:



#### Bits [31:0]

Reserved, RES0.

### D1.2.73 DWT\_PIDR6, DWT Peripheral Identification Register 6

The DWT\_PIDR6 characteristics are:

#### Purpose

Provides CoreSight Unique Component Identifier information for the DWT.

#### Usage constraints

Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

If the Main Extension is not implemented then it is IMPLEMENTATION DEFINED whether this register is accessible only to the debugger and RES0 for software. Otherwise the register is accessible to the debugger and software.

#### Configurations

Present only if CoreSight identification is implemented.

This register is RES0 if CoreSight identification is not implemented.

Present only if the DWT is implemented.

This register is RES0 if the DWT is not implemented.

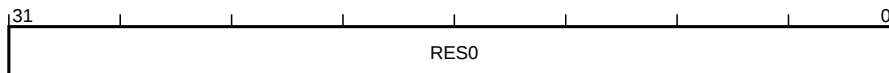
#### Attributes

32-bit read-only register located at 0xE0001FD8.

This register is not banked between Security states.

### Field descriptions

The DWT\_PIDR6 bit assignments are:



#### Bits [31:0]

Reserved, RES0.



### D1.2.74 DWT\_PIDR7, DWT Peripheral Identification Register 7

The DWT\_PIDR7 characteristics are:

#### Purpose

Provides CoreSight Unique Component Identifier information for the DWT.

#### Usage constraints

Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

If the Main Extension is not implemented then it is IMPLEMENTATION DEFINED whether this register is accessible only to the debugger and RES0 for software. Otherwise the register is accessible to the debugger and software.

#### Configurations

Present only if CoreSight identification is implemented.

This register is RES0 if CoreSight identification is not implemented.

Present only if the DWT is implemented.

This register is RES0 if the DWT is not implemented.

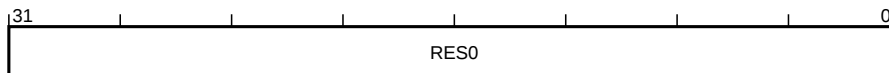
#### Attributes

32-bit read-only register located at 0xE0001FDC.

This register is not banked between Security states.

#### Field descriptions

The DWT\_PIDR7 bit assignments are:



#### Bits [31:0]

Reserved, RES0.

### D1.2.75 DWT\_SLEEPCNT, DWT Sleep Count Register

The DWT\_SLEEPCNT characteristics are:

#### Purpose

Counts the total number of cycles that the processor is sleeping.

#### Usage constraints

Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

#### Configurations

Present only if the Main Extension is implemented.

This register is RES0 if the Main Extension is not implemented.

Present only if the DWT is implemented.

This register is RES0 if the DWT is not implemented.

Present only if `DWT_CTRL.NOPRFCNT == 0`.

This register is RES0 if `DWT_CTRL.NOPRFCNT == 1`.

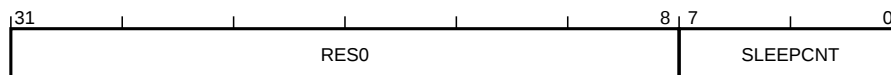
#### Attributes

32-bit read/write register located at `0xE0001010`.

This register is not banked between Security states.

### Field descriptions

The DWT\_SLEEPCNT bit assignments are:



#### Bits [31:8]

Reserved, RES0.

#### SLEEPCNT, bits [7:0]

Sleep counter.

Counts one on each cycle when all of the following are true:

- `DWT_CTRL.SLEEPEVTENA == 1` and `DEMCR.TRCENA == 1`.
- No instruction is executed, see DWT\_CPICNT.
- No load-store operation is in progress, see DWT\_LSUCNT.
- No exception-entry or exception-exit operation is in progress, see DWT\_EXCCNT.
- The PE is in a power-saving mode.
- Either `SecureNoninvasiveDebugAllowed() == TRUE`, or the PE is in Non-secure state and `NoninvasiveDebugAllowed() == TRUE`.

Power-saving modes include WFI, WFE, and Sleep-on-exit.

All power-saving features are IMPLEMENTATION DEFINED and therefore when this counter counts is IMPLEMENTATION DEFINED. In particular, it is IMPLEMENTATION DEFINED whether the counter increments if the PE is in a power-saving mode and `SCR.SLEEPDEEP` is set.

Initialized to zero when the counter is disabled and DWT\_CTRL.SLEEPEVTENA is written with 1. An Event Counter packet is emitted on counter overflow.

This field resets to an **UNKNOWN** value on a Cold reset.

**Note**

Arm recommends that this counter counts all cycles when the PE is sleeping and SCR.SLEEPDEEP is clear, regardless of whether a WFI or WFE instruction, or Sleep-on-exit, caused the entry to the power-saving mode.

## D1.2.76 EPSR, Execution Program Status Register

The EPSR characteristics are:

### Purpose

Holds Execution state bits.

### Usage constraints

Privileged access only. Unprivileged access is RAZ/WI, unless otherwise stated.

### Configurations

This register is always implemented.

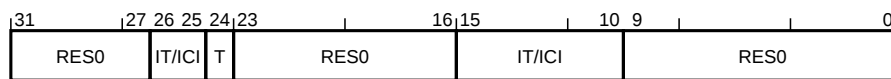
### Attributes

32-bit read/write special-purpose register.

This register is not banked between Security states.

## Field descriptions

The EPSR bit assignments are:



### Bits [31:27]

Reserved, [RES0](#).

### T, bit [24]

T32 state bit. Determines the current instruction set state.

The possible values of this bit are:

**0**

Execution of any instruction generates an INVSTATE UsageFault.

**1**

Instructions decoded as T32 instructions.

This bit resets to an [UNKNOWN](#) value on a Warm reset.

### Bits [23:16]

Reserved, [RES0](#).

### IT/ICI, bits [15:10, 26:25]

If-then and interrupt continuation. Depending on value, this field encodes either the current condition and position in an IT block sequence, or information on the outstanding register list for an interrupted interrupt-continuable multicycle load or store instruction.

The field IT[7:0] is equivalent to EPSR[15:10,26:25]. The field ICI[7:0] is equivalent to EPSR[26:25,15:10].

If the Main Extension is not implemented, this field is [RES0](#).

This field resets to zero on a Warm reset.

### Bits [9:0]

Reserved, [RES0](#).

## D1.2.77 EXC\_RETURN, Exception Return Payload

The EXC\_RETURN characteristics are:

### Purpose

Value provided in LR on entry to an exception, and used with a BX or load to PC to perform an exception return.

### Usage constraints

None.

### Configurations

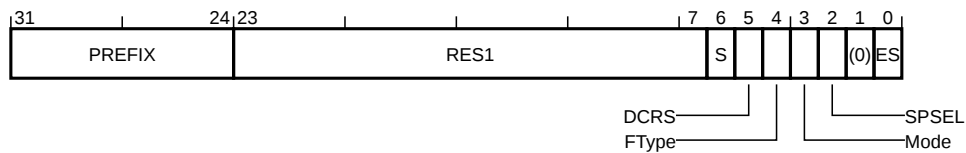
All.

### Attributes

32-bit payload.

## Field descriptions

The EXC\_RETURN bit assignments are:



### PREFIX, bits [31:24]

Prefix. Indicates that this is an EXC\_RETURN value.

This field reads as 0b11111111.

### Bits [23:7]

Reserved, RES1.

### S, bit [6]

Secure or Non-secure stack. Indicates whether a Secure or Non-secure stack is used to restore stack frame on exception return.

The possible values of this bit are:

**0**  
Non-secure stack used.

**1**  
Secure stack used.

If the Security Extension is not implemented, this bit is [UNPREDICTABLE](#).

### DCRS, bit [5]

Default callee register stacking. Indicates whether the default stacking rules apply, or whether the callee registers are already on the stack.

The possible values of this bit are:

**0**  
Stacking of the callee saved registers skipped.

**1**  
Default rules for stacking the callee registers followed.

**FType, bit [4]**

Stack frame type. Indicates whether the stack frame is a standard integer only stack frame or an extended floating-point stack frame.

The possible values of this bit are:

**0**  
Extended stack frame.

**1**  
Standard stack frame.

If the Floating-point Extension is not implemented, this bit is [RES1](#).

**Mode, bit [3]**

Mode. Indicates the Mode that was stacked from.

The possible values of this bit are:

**0**  
Handler mode.

**1**  
Thread mode.

**SPSEL, bit [2]**

Stack pointer selection. The value of this bit indicates the transitory value of the CONTROL.SPSEL bit associated with the Security state of the exception as indicated by EXC\_RETURN.ES.

The possible values of this bit are:

**0**  
Main stack pointer.

**1**  
Process stack pointer.

**Bit [1]**

Reserved, [RES0](#).

**ES, bit [0]**

Exception Secure. The security domain the exception was taken to.

The possible values of this bit are:

**0**  
Non-secure.

**1**  
Secure.

If the Security Extension is not implemented, this bit is [UNPREDICTABLE](#).



### D1.2.79 FNC\_RETURN, Function Return Payload

The FNC\_RETURN characteristics are:

**Purpose**

Value provided in LR on entry to Non-secure state from a Secure BLXNS.

**Usage constraints**

None.

**Configurations**

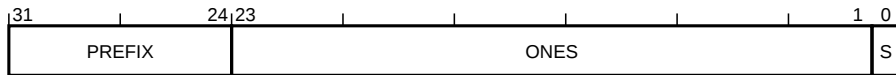
All.

**Attributes**

32-bit payload.

#### Field descriptions

The FNC\_RETURN bit assignments are:



**PREFIX, bits [31:24]**

This field reads as 0b11111110.

**ONES, bits [23:1]**

This field reads as 0b111111111111111111111111.

**S, bit [0]**

Secure. Indicates whether the function call was from the Non-secure or Secure state. Because FNC\_RETURN is only used when calling from the Secure state, this bit is always set to 1. However, some function chaining cases can result in an SG instruction clearing this bit, so the architecture ignores the state of this bit when processing a branch to FNC\_RETURN.

The possible values of this bit are:

**0**  
From Non-secure state.

**1**  
From Secure state.



## D1.2.80 FPCAR, Floating-Point Context Address Register

The FPCAR characteristics are:

### Purpose

Holds the location of the unpopulated floating-point register space allocated on an exception stack frame.

### Usage constraints

Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are **UNPREDICTABLE**.

### Configurations

Present only if the Floating-point Extension is implemented.

This register is **RES0** if the Floating-point Extension is not implemented.

### Attributes

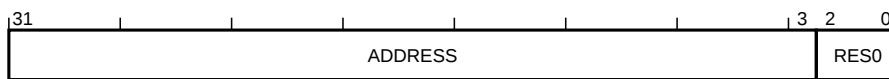
32-bit read/write register located at 0xE000EF38.

Secure software can access the Non-secure version of this register via FPCAR\_NS located at 0xE002EF38. The location 0xE002EF38 is **RES0** to software executing in Non-secure state and the debugger.

This register is banked between Security states.

## Field descriptions

The FPCAR bit assignments are:



### ADDRESS, bits [31:3]

Address. The location of the unpopulated floating-point register space allocated on an exception stack frame.

This field resets to an **UNKNOWN** value on a Warm reset.

### Bits [2:0]

Reserved, **RES0**.

## D1.2.81 FPCCR, Floating-Point Context Control Register

The FPCCR characteristics are:

### Purpose

Holds control data for the Floating Point Unit.

### Usage constraints

Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are **UNPREDICTABLE**.

### Configurations

Present only if the Floating-point Extension is implemented.

This register is **RES0** if the Floating-point Extension is not implemented.

### Attributes

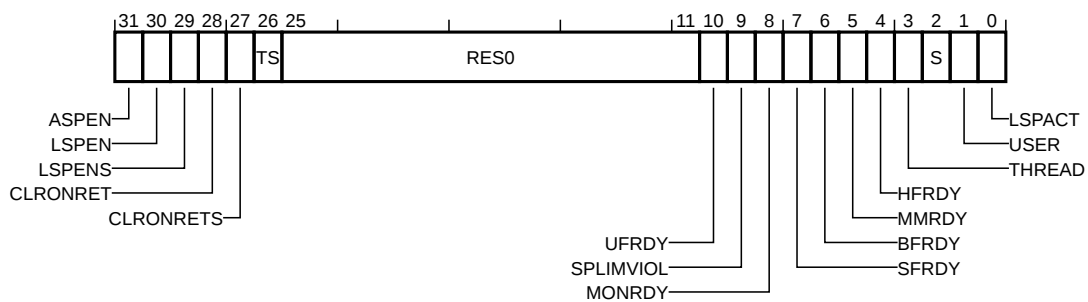
32-bit read/write register located at 0xE000EF34.

Secure software can access the Non-secure version of this register via FPCCR\_NS located at 0xE002EF34. The location 0xE002EF34 is **RES0** to software executing in Non-secure state and the debugger.

This register is not banked between Security states.

## Field descriptions

The FPCCR bit assignments are:



### ASPEN, bit [31]

Automatic state preservation enable. When this bit is set to 1, execution of a floating-point instruction sets the CONTROL.FPCA bit to 1.

This bit is banked between Security states.

The possible values of this bit are:

**0**

Executing an FP instruction has no effect on CONTROL.FPCA.

**1**

Executing an FP instruction sets CONTROL.FPCA to 1.

Setting this bit to 1 means the hardware automatically preserves floating-point context on exception entry and restores it on exception return.

This bit resets to one on a Warm reset.

### LSPEN, bit [30]

Lazy state preservation enable. Enables lazy context save of floating-point state.

The possible values of this bit are:

**0**  
Disable automatic lazy context save.

**1**  
Enable automatic lazy context save.

Writes to this bit from Non-secure state are ignored if LSPENS is set to 1.

This bit resets to one on a Warm reset.

**LSPENS, bit [29]**

Lazy state preservation enable Secure. This bit controls whether the LSPEN bit is writable from the Non-secure state. This behaves as RAZ/WI when accessed from the Non-secure state.

The possible values of this bit are:

**0**  
LSPEN is readable and writable from both Security states.

**1**  
LSPEN is readable from both Security states, but writes to LSPEN are ignored from the Non-secure state.

This bit is RAZ/WI from Non-secure state.

This bit resets to zero on a Warm reset.

**CLRONRET, bit [28]**

Clear on return. Clear floating-point caller saved registers on exception return.

The possible values of this bit are:

**0**  
Disabled.

**1**  
Enabled.

When set to 1 the caller saved floating-point registers S0 to S15, and FPSCR are cleared on exception return (including tail chaining) if CONTROL.FPCA is set to 1 and FPCCR\_S.LSPACT is set to 0. Writes to this bit from Non-secure state are ignored if CLRONRETS is set to one.

This bit resets to zero on a Warm reset.

**CLRONRETS, bit [27]**

Clear on return, Secure only. This bit controls whether the CLRONRET bit is writable from the Non-secure state.

The possible values of this bit are:

**0**  
The CLRONRET field is accessible from both Security states.

**1**  
The Non-secure view of the CLRONRET field is read-only.

This bit is RAZ/WI from Non-secure state.

This bit resets to zero on a Warm reset.

**TS, bit [26]**

Treat as Secure. Treat floating-point registers as Secure enable.

The possible values of this bit are:

**0**  
Disabled.

**1**

Enabled.

When set to 0 the floating-point registers are treated as Non-secure even when the PE is in Secure state and, therefore, the callee saved registers are never pushed to the stack. If the floating-point registers never contain data that needs to be protected, clearing this flag can reduce interrupt latency. As this field changes how secure stack frames are interpreted, **UNPREDICTABLE** behavior can result if the state of this bit is not consistent with the current Secure stacks. Therefore, firmware must take care when modifying this value. This field behaves as RAZ/WI from the Non-secure state.

This bit is RAZ/WI from Non-secure state.

This bit resets to zero on a Warm reset.

**Bits [25:11]**

Reserved, **RES0**.

**UFRDY, bit [10]**

UsageFault enable. Indicates whether the software executing, when the PE allocated the floating-point stack frame, was able to set the UsageFault exception to pending.

This bit is banked between Security states.

The possible values of this bit are:

**0**

Not able to set the UsageFault exception to pending.

**1**

Able to set the UsageFault exception to pending.

This bit resets to an **UNKNOWN** value on a Warm reset.

**SPLIMVIOL, bit [9]**

Stack pointer limit violation. This bit indicates whether the floating-point context violates the stack pointer limit that was active when lazy state preservation was activated. SPLIMVIOL modifies the lazy floating-point state preservation behavior.

This bit is banked between Security states.

The possible values of this bit are:

**0**

The existing behavior is retained.

**1**

The memory accesses associated with the floating-point state preservation are not performed. However if the floating-point state is Secure and FPCCR.TS is set to 1 the registers are still zeroed and the floating-point state is lost.

This bit resets to an **UNKNOWN** value on a Warm reset.

**MONRDY, bit [8]**

DebugMonitor ready. Indicates whether the software executing, when the PE allocated the floating-point stack frame, was able to set the DebugMonitor exception to pending.

The possible values of this bit are:

**0**

Not able to set the DebugMonitor exception to pending.

**1**

Able to set the DebugMonitor exception to pending.

If DEMCR.SDME is one this bit is RAZ/WI from Non-secure state

This bit resets to an **UNKNOWN** value on a Warm reset.

**SFRDY, bit [7]**

SecureFault ready. Indicates whether the software executing, when the PE allocated the floating-point stack frame, was able to set the SecureFault exception to pending.

This bit is RAZ/WI from Non-secure state.

This bit resets to an **UNKNOWN** value on a Warm reset.

**BFRDY, bit [6]**

BusFault ready. Indicates whether the software executing, when the PE allocated the floating-point stack frame, was able to set the BusFault exception to pending.

The possible values of this bit are:

**0**

Not able to set the BusFault exception to pending.

**1**

Able to set the BusFault exception to pending.

If AIRCR.BFHFNMINS is zero this bit is RAZ/WI from Non-secure state.

This bit resets to an **UNKNOWN** value on a Warm reset.

**MMRDY, bit [5]**

MemManage ready. Indicates whether the software executing, when the PE allocated the floating-point stack frame, was able to set the MemManage exception to pending.

This bit is banked between Security states.

The possible values of this bit are:

**0**

Not able to set the MemManage exception to pending.

**1**

Able to set the MemManage exception to pending.

This bit resets to an **UNKNOWN** value on a Warm reset.

**HFRDY, bit [4]**

HardFault ready. Indicates whether the software executing, when the PE allocated the floating-point stack frame, was able to set the HardFault exception to pending.

The possible values of this bit are:

**0**

Not able to set the HardFault exception to pending.

**1**

Able to set the HardFault exception to pending.

If AIRCR.BFHFNMINS is zero this bit is RAZ/WI from Non-secure state.

This bit resets to an **UNKNOWN** value on a Warm reset.

**THREAD, bit [3]**

Thread mode. Indicates the PE mode when it allocated the floating-point stack frame.

This bit is banked between Security states.

The possible values of this bit are:

**0**  
Handler mode.

**1**  
Thread mode.

This bit is for fault handler information only and does not interact with the exception model.

This bit resets to an **UNKNOWN** value on a Warm reset.

**S, bit [2]**

Security. Security status of the floating-point context. This bit is only present in the Secure version of the register. This bit is updated whenever lazy state preservation is activated, or when a floating-point instruction is executed.

The possible values of this bit are:

**0**  
Indicates the floating-point context belongs to the Non-secure state.

**1**  
Indicates the floating-point context belongs to the Secure state.

This bit is RAZ/WI from Non-secure state.

This bit resets to one on a Warm reset.

**USER, bit [1]**

User privilege. Indicates the privilege level of the software executing when the PE allocated the floating-point stack frame.

This bit is banked between Security states.

The possible values of this bit are:

**0**  
Privileged.

**1**  
Unprivileged.

This bit resets to an **UNKNOWN** value on a Warm reset.

**LSPACT, bit [0]**

Lazy state preservation active. Indicates whether lazy preservation of the floating-point state is active.

This bit is banked between Security states.

The possible values of this bit are:

**0**  
Lazy state preservation is not active.

**1**  
Lazy state preservation is active.

This bit resets to zero on a Warm reset.

## D1.2.82 FPDSCR, Floating-Point Default Status Control Register

The FPDSCR characteristics are:

### Purpose

Holds the default values for the floating-point status control data that the PE assigns to FPSCR when it creates a new floating-point context.

### Usage constraints

Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are **UNPREDICTABLE**.

### Configurations

Present only if the Floating-point Extension is implemented.

This register is **RES0** if the Floating-point Extension is not implemented.

### Attributes

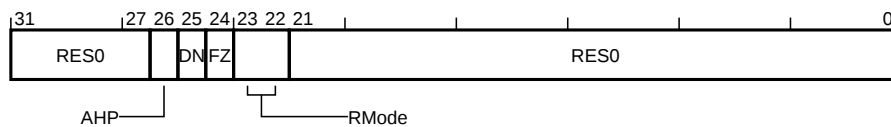
32-bit read/write register located at 0xE000EF3C.

Secure software can access the Non-secure version of this register via FPDSCR\_NS located at 0xE002EF3C. The location 0xE002EF3C is **RES0** to software executing in Non-secure state and the debugger.

This register is banked between Security states.

## Field descriptions

The FPDSCR bit assignments are:



### Bits [31:27]

Reserved, **RES0**.

### AHP, bit [26]

Alternative half-precision. Default value for FPSCR.AHP.

This bit resets to zero on a Warm reset.

### DN, bit [25]

Default NaN. Default value for FPSCR.DN.

This bit resets to zero on a Warm reset.

### FZ, bit [24]

Flush-to-zero. Default value for FPSCR.FZ.

This bit resets to zero on a Warm reset.

### RMode, bits [23:22]

Rounding mode. Default value for FPSCR.RMode.

This field resets to zero on a Warm reset.

### Bits [21:0]

Reserved, **RES0**.

### D1.2.83 FPSCR, Floating-point Status and Control Register

The FPSCR characteristics are:

**Purpose**

Provides control of the Floating-point unit.

**Usage constraints**

Privileged access only. Unprivileged access is RAZ/WI, unless otherwise stated.

**Configurations**

Present only if the Floating-point Extension is implemented.

This register is [RES0](#) if the Floating-point Extension is not implemented.

**Attributes**

32-bit read/write special-purpose register.

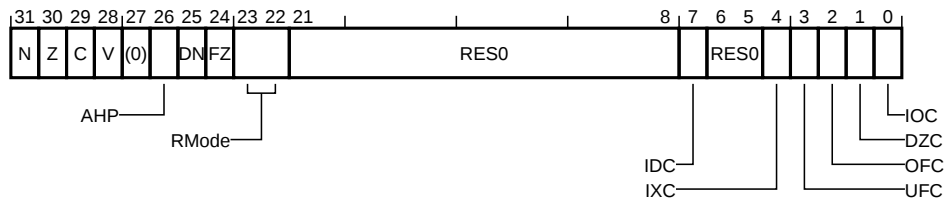
This register is not banked between Security states.

### Preface

Writes to FPSCR can have side-effects on various aspects of processor operation. All of these side-effects are synchronous to FPSCR write. This means that they are guaranteed not to be visible to earlier instructions in the execution stream, and they are guaranteed to be visible to later instructions in the execution stream.

### Field descriptions

The FPSCR bit assignments are:



**N, bit [31]**

Negative condition flag. When updated by a VCMP instruction, this bit indicates whether the result was less than.

The possible values of this bit are:

**0**  
Compare result was not less than.

**1**  
Compare result was less than.

See VCMP for details.

This bit resets to an [UNKNOWN](#) value on a Warm reset.

**Z, bit [30]**

Zero condition flag. When updated by a VCMP instruction, this bit indicates whether the result was equal.

The possible values of this bit are:

**0**  
Compare result was not equal.



**1**

Compare result was equal.

See VCMP for details.

This bit resets to an **UNKNOWN** value on a Warm reset.

**C, bit [29]**

Carry condition flag. When updated by a VCMP instruction, this bit indicates whether the result was not less than.

The possible values of this bit are:

**0**

Compare result was less than.

**1**

Compare result was not less than.

See VCMP for details.

This bit resets to an **UNKNOWN** value on a Warm reset.

**V, bit [28]**

Overflow condition flag. When updated by a VCMP instruction, this bit indicates whether the result was unordered.

The possible values of this bit are:

**0**

Compare result was not unordered.

**1**

Compare result was unordered.

See VCMP for details.

This bit resets to an **UNKNOWN** value on a Warm reset.

**Bit [27]**

Reserved, **RES0**.

**AHP, bit [26]**

Alternative half-precision control bit. This bit controls how the PE interprets 16-bit floating-point values.

The possible values of this bit are:

**0**

IEEE half-precision format selected.

**1**

Alternative half-precision format selected.

This bit resets to an **UNKNOWN** value on a Warm reset.

**DN, bit [25]**

Default NaN mode control bit. This bit determines whether floating-point operations propagate NaNs or use the Default NaN.

The possible values of this bit are:

**0**

NaN operands propagate through to the output of a floating-point operation.

**1**

Any operation involving one or more NaNs returns the Default NaN.

This bit resets to an **UNKNOWN** value on a Warm reset.

**FZ, bit [24]**

Flush-to-zero mode control. This bit determines whether denormal floating-point values are treated as though zero.

The possible values of this bit are:

**0**

Flush-to-zero mode disabled. Behavior of the Floating-point unit is fully compliant with the IEEE754 standard.

**1**

Flush-to-zero mode enabled.

This bit resets to an **UNKNOWN** value on a Warm reset.

**RMode, bits [23:22]**

Rounding mode control field. This field determines what rounding mode is applied to floating-point operations.

The possible values of this field are:

**0b00**

Round to Nearest (RN) mode.

**0b01**

Round towards Plus Infinity (RP) mode.

**0b10**

Round towards Minus Infinity (RM) mode.

**0b11**

Round towards Zero (RZ) mode.

This field resets to an **UNKNOWN** value on a Warm reset.

**Bits [21:8]**

Reserved, **RES0**.

**IDC, bit [7]**

Input Denormal cumulative exception bit. This sticky flag records whether a floating-point input denormal exception has been detected since last cleared.

The possible values of this bit are:

**0**

Input Denormal exception has not occurred since 0 was last written to this bit.

**1**

Input Denormal exception has occurred since 0 was last written to this bit.

This bit resets to an **UNKNOWN** value on a Warm reset.

**Bits [6:5]**

Reserved, **RES0**.

**IXC, bit [4]**

Inexact cumulative exception bit. This sticky flag records whether a floating-point inexact exception has been detected since last cleared.

The possible values of this bit are:

**0**

Inexact exception has not occurred since 0 was last written to this bit.

**1**

Inexact exception has occurred since 0 was last written to this bit.

This bit resets to an **UNKNOWN** value on a Warm reset.

**UFC, bit [3]**

Underflow cumulative exception bit. This sticky flag records whether a floating-point Underflow exception has been detected since last cleared.

The possible values of this bit are:

**0**

Underflow exception has not occurred since 0 was last written to this bit.

**1**

Underflow exception has occurred since 0 was last written to this bit.

**OFC, bit [2]**

Overflow cumulative exception bit. This sticky flag records whether a floating-point overflow exception has been detected since last cleared.

The possible values of this bit are:

**0**

Overflow exception has not occurred since 0 was last written to this bit.

**1**

Overflow exception has occurred since 0 was last written to this bit.

This bit resets to an **UNKNOWN** value on a Warm reset.

**DZC, bit [1]**

Divide by Zero cumulative exception bit. This sticky flag records whether a floating-point divide by zero exception has been detected since last cleared.

The possible values of this bit are:

**0**

Division by Zero exception has not occurred since 0 was last written to this bit.

**1**

Division by Zero exception has occurred since 0 was last written to this bit.

This bit resets to an **UNKNOWN** value on a Warm reset.

**IOC, bit [0]**

Invalid Operation cumulative exception bit. This sticky flag records whether a floating-point invalid operation exception has been detected since last cleared.

The possible values of this bit are:

**0**

Invalid Operation exception has not occurred since 0 was last written to this bit.

**1**

Invalid Operation exception has occurred since 0 was last written to this bit.

This bit resets to an **UNKNOWN** value on a Warm reset.

## D1.2.84 FP\_CIDR0, FP Component Identification Register 0

The FP\_CIDR0 characteristics are:

### Purpose

Provides CoreSight Unique Component Identifier information for the Flash Patch and Breakpoint Unit.

### Usage constraints

Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are **UNPREDICTABLE**.

If the Main Extension is not implemented then it is **IMPLEMENTATION DEFINED** whether this register is accessible only to the debugger and **RES0** for software. Otherwise the register is accessible to the debugger and software.

### Configurations

Present only if CoreSight identification is implemented.

This register is **RES0** if CoreSight identification is not implemented.

Present only if the FPB is implemented.

This register is **RES0** if the FPB is not implemented.

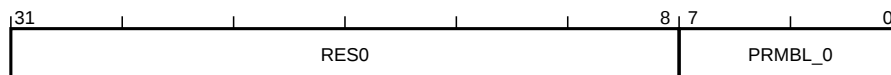
### Attributes

32-bit read-only register located at 0xE0002FF0.

This register is not banked between Security states.

## Field descriptions

The FP\_CIDR0 bit assignments are:



### Bits [31:8]

Reserved, **RES0**.

### PRMBL\_0, bits [7:0]

CoreSight component identification preamble. See the *Arm® CoreSight™ Architecture Specification*.

This field reads as 0x0D.



## D1.2.86 FP\_CIDR2, FP Component Identification Register 2

The FP\_CIDR2 characteristics are:

### Purpose

Provides CoreSight Unique Component Identifier information for the Flash Patch and Breakpoint Unit.

### Usage constraints

Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are **UNPREDICTABLE**.

If the Main Extension is not implemented then it is **IMPLEMENTATION DEFINED** whether this register is accessible only to the debugger and **RES0** for software. Otherwise the register is accessible to the debugger and software.

### Configurations

Present only if CoreSight identification is implemented.

This register is **RES0** if CoreSight identification is not implemented.

Present only if the FPB is implemented.

This register is **RES0** if the FPB is not implemented.

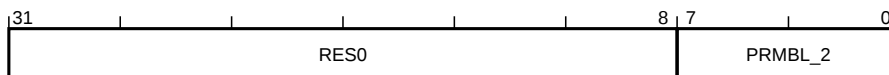
### Attributes

32-bit read-only register located at 0xE0002FF8.

This register is not banked between Security states.

## Field descriptions

The FP\_CIDR2 bit assignments are:



### Bits [31:8]

Reserved, **RES0**.

### PRMBL\_2, bits [7:0]

CoreSight component identification preamble. See the *Arm® CoreSight™ Architecture Specification*.

This field reads as 0x05.

### D1.2.87 FP\_CIDR3, FP Component Identification Register 3

The FP\_CIDR3 characteristics are:

#### Purpose

Provides CoreSight Unique Component Identifier information for the Flash Patch and Breakpoint Unit.

#### Usage constraints

Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are **UNPREDICTABLE**.

If the Main Extension is not implemented then it is **IMPLEMENTATION DEFINED** whether this register is accessible only to the debugger and **RES0** for software. Otherwise the register is accessible to the debugger and software.

#### Configurations

Present only if CoreSight identification is implemented.

This register is **RES0** if CoreSight identification is not implemented.

Present only if the FPB is implemented.

This register is **RES0** if the FPB is not implemented.

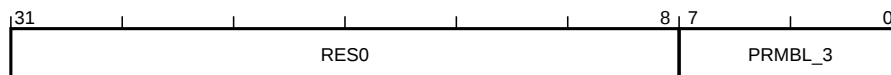
#### Attributes

32-bit read-only register located at 0xE0002FFC.

This register is not banked between Security states.

### Field descriptions

The FP\_CIDR3 bit assignments are:



#### Bits [31:8]

Reserved, **RES0**.

#### PRMBL\_3, bits [7:0]

CoreSight component identification preamble. See the *Arm® CoreSight™ Architecture Specification*.

This field reads as 0xB1.

## D1.2.88 FP\_COMPn, Flash Patch Comparator Register, n = 0 - 125

The FP\_COMP{0..125} characteristics are:

### Purpose

Holds an address for comparison.

### Usage constraints

Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

If the Main Extension is not implemented then it is IMPLEMENTATION DEFINED whether this register is accessible only to the debugger and RES0 for software. Otherwise the register is accessible to the debugger and software.

### Configurations

Present only if the FPB is implemented.

This register is RES0 if the FPB is not implemented.

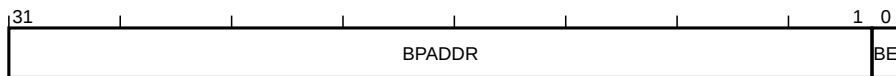
### Attributes

32-bit read/write register located at  $0 \times E0002008 + 4n$ .

This register is not banked between Security states.

## Field descriptions

The FP\_COMP{0..125} bit assignments are:



### BPADDR, bits [31:1]

Breakpoint address. Specifies bits[31:1] of the breakpoint instruction address.

### BE, bit [0]

Breakpoint enable. Selects between remapping and breakpoint functionality.

The possible values of this bit are:

**0**

Breakpoint disabled.

**1**

Breakpoint enabled.

For backwards compatibility, when disabling a breakpoint software must write zero to the whole register.

This bit resets to zero on a Cold reset.



## D1.2.89 FP\_CTRL, Flash Patch Control Register

The FP\_CTRL characteristics are:

### Purpose

Provides FPB implementation information, and the global enable for the FPB unit.

### Usage constraints

Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are **UNPREDICTABLE**.

If the Main Extension is not implemented then it is **IMPLEMENTATION DEFINED** whether this register is accessible only to the debugger and **RES0** for software. Otherwise the register is accessible to the debugger and software.

### Configurations

Present only if the FPB is implemented.

This register is **RES0** if the FPB is not implemented.

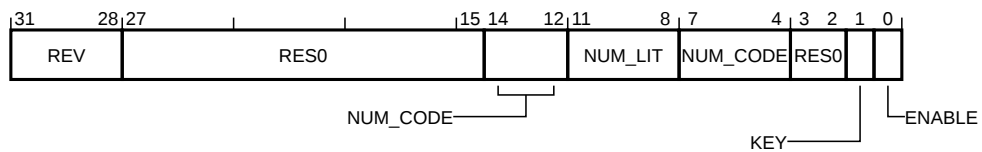
### Attributes

32-bit read/write register located at 0xE0002000.

This register is not banked between Security states.

## Field descriptions

The FP\_CTRL bit assignments are:



### REV, bits [31:28]

Revision. Flash Patch and Breakpoint Unit architecture revision.

The possible values of this field are:

#### 0b0001

Flash Patch Breakpoint version 2 implemented.

All other values are reserved.

This field is read-only.

This field reads as 0b0001.

### Bits [27:15]

Reserved, **RES0**.

### NUM\_CODE, bits [14:12,7:4]

Number of implemented code comparators. Indicates the number of implemented instruction address comparators. Zero indicates no Instruction Address comparators are implemented. The Instruction Address comparators are numbered from 0 to NUM\_CODE - 1.

This field is read-only.

This field reads as an **IMPLEMENTATION DEFINED** value.

**NUM\_LIT, bits [11:8]**

Number of literal comparators. This field is RAZ/WI. Remapping is not supported in Armv8-M.

**Bits [3:2]**

Reserved, **RES0**.

**KEY, bit [1]**

FP\_CTRL write-enable key. Writes to the FP\_CTRL are ignored unless KEY is concurrently written to one.

The possible values of this bit are:

**0**

Concurrent write to FP\_CTRL ignored.

**1**

Concurrent write to FP\_CTRL permitted.

This bit reads-as-zero.

**ENABLE, bit [0]**

Flash Patch global enable. Enables the FPB.

The possible values of this bit are:

**0**

All FPB functionality disabled.

**1**

FPB enabled.

This bit resets to zero on a Cold reset.

## D1.2.90 FP\_DEVARCH, FPB Device Architecture Register

The FP\_DEVARCH characteristics are:

### Purpose

Provides CoreSight Unique Component Identifier information for the Flash Patch and Breakpoint Unit.

### Usage constraints

Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are **UNPREDICTABLE**.

If the Main Extension is not implemented then it is **IMPLEMENTATION DEFINED** whether this register is accessible only to the debugger and **RES0** for software. Otherwise the register is accessible to the debugger and software.

### Configurations

Present only if CoreSight identification is implemented.

This register is **RES0** if CoreSight identification is not implemented.

Present only if the FPB is implemented.

This register is **RES0** if the FPB is not implemented.

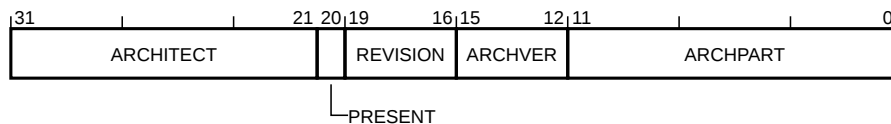
### Attributes

32-bit read-only register located at 0xE0002FBC.

This register is not banked between Security states.

## Field descriptions

The FP\_DEVARCH bit assignments are:



### ARCHITECT, bits [31:21]

Architect. Defines the architect of the component. Bits [31:28] are the JEP106 continuation code (JEP106 bank ID, minus 1) and bits [27:21] are the JEP106 ID code.

The possible values of this field are:

#### 0x23B

JEP106 continuation code 0x4, ID code 0x3B. Arm Limited.

Other values are defined by the JEDEC JEP106 standard.

This field reads as 0x23B.

### PRESENT, bit [20]

DEVARCH Present. Defines that the DEVARCH register is present.

The possible values of this bit are:

#### 1

DEVARCH information present.

This bit reads as one.

**REVISION, bits [19:16]**

Revision. Defines the architecture revision of the component.

The possible values of this field are:

**0b0000**

FPB architecture v2.0.

This field reads as 0b0000.

**ARCHVER, bits [15:12]**

Architecture Version. Defines the architecture version of the component.

The possible values of this field are:

**0b0001**

FPB architecture v2.

ARCHVER and ARCHPART are also defined as a single field, ARCHID, so that ARCHVER is ARCHID[15:12].

This field reads as 0b0001.

**ARCHPART, bits [11:0]**

Architecture Part. Defines the architecture of the component.

The possible values of this field are:

**0xA03**

FPB architecture.

ARCHVER and ARCHPART are also defined as a single field, ARCHID, so that ARCHPART is ARCHID[11:0].

This field reads as 0xA03.

## D1.2.91 FP\_DEVTYPE, FPB Device Type Register

The FP\_DEVTYPE characteristics are:

### Purpose

Provides CoreSight Unique Component Identifier information for the Flash Patch and Breakpoint Unit.

### Usage constraints

Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are **UNPREDICTABLE**.

If the Main Extension is not implemented then it is **IMPLEMENTATION DEFINED** whether this register is accessible only to the debugger and **RES0** for software. Otherwise the register is accessible to the debugger and software.

### Configurations

Present only if CoreSight identification is implemented.

This register is **RES0** if CoreSight identification is not implemented.

Present only if the FPB is implemented.

This register is **RES0** if the FPB is not implemented.

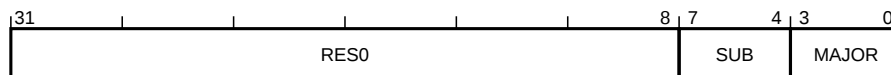
### Attributes

32-bit read-only register located at 0xE0002FCC.

This register is not banked between Security states.

## Field descriptions

The FP\_DEVTYPE bit assignments are:



### Bits [31:8]

Reserved, **RES0**.

### SUB, bits [7:4]

Sub-type. Component sub-type.

The possible values of this field are:

**0x0**

Other.

This field reads as 0b0000.

### MAJOR, bits [3:0]

Major type. Component major type.

The possible values of this field are:

**0x0**

Miscellaneous.

This field reads as 0b0000.

## D1.2.92 FP\_LAR, FPB Software Lock Access Register

The FP\_LAR characteristics are:

### Purpose

Provides CoreSight Software Lock control for the FPB, see the *Arm® CoreSight™ Architecture Specification* for details.

### Usage constraints

Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are **UNPREDICTABLE**.

This register is RAZ/WI if accessed via the debugger.

### Configurations

Present only if CoreSight identification is implemented.

This register is **RES0** if CoreSight identification is not implemented.

Present only if the FPB is implemented.

This register is **RES0** if the FPB is not implemented.

Present only if the optional Software Lock is implemented.

This register is RAZ/WI if the Software Lock is not implemented.

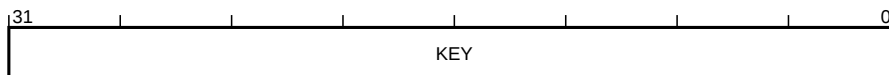
### Attributes

32-bit write-only register located at 0xE0002FB0.

This register is not banked between Security states.

## Field descriptions

The FP\_LAR bit assignments are:



### KEY, bits [31:0]

Lock Access control.

Writing the key value 0xC5ACCE55 to this field unlocks the lock, enabling write accesses to the registers of this component through a memory-mapped interface.

Writing any other value to this register locks the lock, disabling write accesses to the registers of this component through a memory mapped interface.

### D1.2.93 FP\_LSR, FPB Software Lock Status Register

The FP\_LSR characteristics are:

#### Purpose

Provides CoreSight Software Lock status information for the FPB, see the *Arm® CoreSight™ Architecture Specification* for details.

#### Usage constraints

Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

This register is RAZ/WI if accessed via the debugger.

#### Configurations

Present only if CoreSight identification is implemented.

This register is RES0 if CoreSight identification is not implemented.

Present only if the FPB is implemented.

This register is RES0 if the FPB is not implemented.

Present only if the optional Software Lock is implemented.

This register is RAZ/WI if the Software Lock is not implemented.

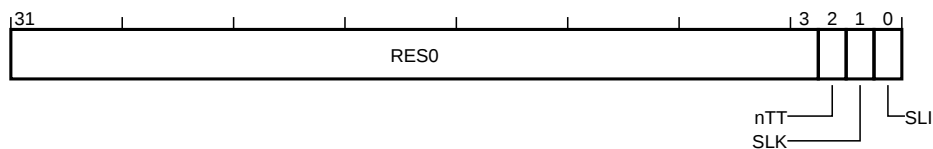
#### Attributes

32-bit read-only register located at 0xE0002FB4.

This register is not banked between Security states.

### Field descriptions

The FP\_LSR bit assignments are:



#### Bits [31:3]

Reserved, RES0.

#### nTT, bit [2]

Not thirty-two bit. See the *Arm® CoreSight™ Architecture Specification*.

This bit reads as zero.

#### SLK, bit [1]

Software Lock status. See the *Arm® CoreSight™ Architecture Specification*.

The possible values of this bit are:

**0**

Lock clear. Software writes are permitted to the registers of this component.

**1**

Lock set. Software writes to the registers of this component are ignored, and reads have no side-effects.

For a debugger read of this register, or when the Software Lock is not implemented, this bit is **RES0**.

This bit resets to one on a Cold reset.

**SLI, bit [0]**

Software Lock implemented. See the *Arm® CoreSight™ Architecture Specification*.

The possible values of this bit are:

**0**

Software Lock not implemented or debugger access.

**1**

Software Lock is implemented and software access.

For a debugger read of this register, or when the Software Lock is not implemented, this bit is **RAZ**.

This bit reads as an **IMPLEMENTATION DEFINED** value.



## D1.2.94 FP\_PIDR0, FP Peripheral Identification Register 0

The FP\_PIDR0 characteristics are:

### Purpose

Provides CoreSight Unique Component Identifier information for the Flash Patch and Breakpoint Unit.

### Usage constraints

Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are **UNPREDICTABLE**.

If the Main Extension is not implemented then it is **IMPLEMENTATION DEFINED** whether this register is accessible only to the debugger and **RES0** for software. Otherwise the register is accessible to the debugger and software.

### Configurations

Present only if CoreSight identification is implemented.

This register is **RES0** if CoreSight identification is not implemented.

Present only if the FPB is implemented.

This register is **RES0** if the FPB is not implemented.

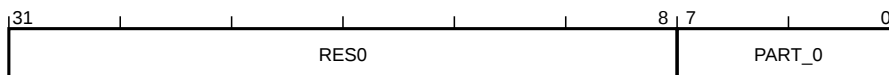
### Attributes

32-bit read-only register located at 0xE0002FE0.

This register is not banked between Security states.

## Field descriptions

The FP\_PIDR0 bit assignments are:



### Bits [31:8]

Reserved, **RES0**.

### PART\_0, bits [7:0]

Part number bits [7:0]. See the *Arm® CoreSight™ Architecture Specification*.

This field reads as an **IMPLEMENTATION DEFINED** value.



## D1.2.96 FP\_PIDR2, FP Peripheral Identification Register 2

The FP\_PIDR2 characteristics are:

### Purpose

Provides CoreSight Unique Component Identifier information for the Flash Patch and Breakpoint Unit.

### Usage constraints

Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are **UNPREDICTABLE**.

If the Main Extension is not implemented then it is **IMPLEMENTATION DEFINED** whether this register is accessible only to the debugger and **RES0** for software. Otherwise the register is accessible to the debugger and software.

### Configurations

Present only if CoreSight identification is implemented.

This register is **RES0** if CoreSight identification is not implemented.

Present only if the FPB is implemented.

This register is **RES0** if the FPB is not implemented.

### Attributes

32-bit read-only register located at 0xE0002FE8.

This register is not banked between Security states.

## Field descriptions

The FP\_PIDR2 bit assignments are:



### Bits [31:8]

Reserved, **RES0**.

### REVISION, bits [7:4]

Component revision. See the *Arm® CoreSight™ Architecture Specification*.

This field reads as an **IMPLEMENTATION DEFINED** value.

### JEDEC, bit [3]

JEDEC assignee value is used. See the *Arm® CoreSight™ Architecture Specification*.

This bit reads as one.

### DES\_1, bits [2:0]

JEP106 identification code bits [6:4]. See the *Arm® CoreSight™ Architecture Specification*.

This field reads as an **IMPLEMENTATION DEFINED** value.

## D1.2.97 FP\_PIDR3, FP Peripheral Identification Register 3

The FP\_PIDR3 characteristics are:

### Purpose

Provides CoreSight Unique Component Identifier information for the Flash Patch and Breakpoint Unit.

### Usage constraints

Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are **UNPREDICTABLE**.

If the Main Extension is not implemented then it is **IMPLEMENTATION DEFINED** whether this register is accessible only to the debugger and **RES0** for software. Otherwise the register is accessible to the debugger and software.

### Configurations

Present only if CoreSight identification is implemented.

This register is **RES0** if CoreSight identification is not implemented.

Present only if the FPB is implemented.

This register is **RES0** if the FPB is not implemented.

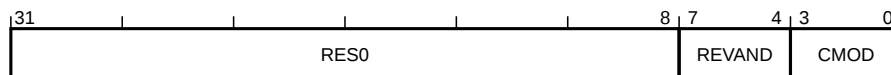
### Attributes

32-bit read-only register located at 0xE0002FEC.

This register is not banked between Security states.

## Field descriptions

The FP\_PIDR3 bit assignments are:



### Bits [31:8]

Reserved, **RES0**.

### REVAND, bits [7:4]

RevAnd. See the *Arm® CoreSight™ Architecture Specification*.

This field reads as an **IMPLEMENTATION DEFINED** value.

### CMOD, bits [3:0]

Customer Modified. See the *Arm® CoreSight™ Architecture Specification*.

This field reads as an **IMPLEMENTATION DEFINED** value.

## D1.2.98 FP\_PIDR4, FP Peripheral Identification Register 4

The FP\_PIDR4 characteristics are:

### Purpose

Provides CoreSight Unique Component Identifier information for the Flash Patch and Breakpoint Unit.

### Usage constraints

Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are **UNPREDICTABLE**.

If the Main Extension is not implemented then it is **IMPLEMENTATION DEFINED** whether this register is accessible only to the debugger and **RES0** for software. Otherwise the register is accessible to the debugger and software.

### Configurations

Present only if CoreSight identification is implemented.

This register is **RES0** if CoreSight identification is not implemented.

Present only if the FPB is implemented.

This register is **RES0** if the FPB is not implemented.

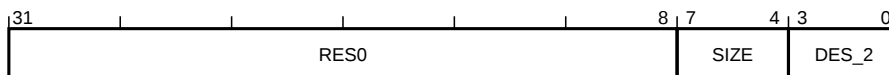
### Attributes

32-bit read-only register located at 0xE0002FD0.

This register is not banked between Security states.

## Field descriptions

The FP\_PIDR4 bit assignments are:



### Bits [31:8]

Reserved, **RES0**.

### SIZE, bits [7:4]

4KB count. See the *Arm® CoreSight™ Architecture Specification*.

This field reads as zero.

### DES\_2, bits [3:0]

JEP106 continuation code. See the *Arm® CoreSight™ Architecture Specification*.

This field reads as an **IMPLEMENTATION DEFINED** value.

## D1.2.99 FP\_PIDR5, FP Peripheral Identification Register 5

The FP\_PIDR5 characteristics are:

### Purpose

Provides CoreSight Unique Component Identifier information for the Flash Patch and Breakpoint Unit.

### Usage constraints

Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

If the Main Extension is not implemented then it is IMPLEMENTATION DEFINED whether this register is accessible only to the debugger and RES0 for software. Otherwise the register is accessible to the debugger and software.

### Configurations

Present only if CoreSight identification is implemented.

This register is RES0 if CoreSight identification is not implemented.

Present only if the FPB is implemented.

This register is RES0 if the FPB is not implemented.

### Attributes

32-bit read-only register located at 0xE0002FD4.

This register is not banked between Security states.

## Field descriptions

The FP\_PIDR5 bit assignments are:



### Bits [31:0]

Reserved, RES0.

### D1.2.100 FP\_PIDR6, FP Peripheral Identification Register 6

The FP\_PIDR6 characteristics are:

#### Purpose

Provides CoreSight Unique Component Identifier information for the Flash Patch and Breakpoint Unit.

#### Usage constraints

Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

If the Main Extension is not implemented then it is IMPLEMENTATION DEFINED whether this register is accessible only to the debugger and RES0 for software. Otherwise the register is accessible to the debugger and software.

#### Configurations

Present only if CoreSight identification is implemented.

This register is RES0 if CoreSight identification is not implemented.

Present only if the FPB is implemented.

This register is RES0 if the FPB is not implemented.

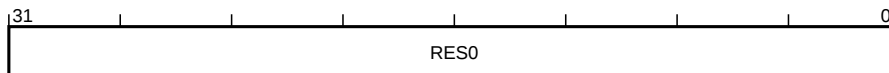
#### Attributes

32-bit read-only register located at 0xE0002FD8.

This register is not banked between Security states.

#### Field descriptions

The FP\_PIDR6 bit assignments are:



#### Bits [31:0]

Reserved, RES0.

### D1.2.101 FP\_PIDR7, FP Peripheral Identification Register 7

The FP\_PIDR7 characteristics are:

#### Purpose

Provides CoreSight Unique Component Identifier information for the Flash Patch and Breakpoint Unit.

#### Usage constraints

Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

If the Main Extension is not implemented then it is IMPLEMENTATION DEFINED whether this register is accessible only to the debugger and RES0 for software. Otherwise the register is accessible to the debugger and software.

#### Configurations

Present only if CoreSight identification is implemented.

This register is RES0 if CoreSight identification is not implemented.

Present only if the FPB is implemented.

This register is RES0 if the FPB is not implemented.

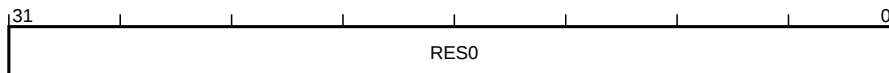
#### Attributes

32-bit read-only register located at 0xE0002FDC.

This register is not banked between Security states.

#### Field descriptions

The FP\_PIDR7 bit assignments are:



#### Bits [31:0]

Reserved, RES0.



## D1.2.102 FP\_REMAP, Flash Patch Remap Register

The FP\_REMAP characteristics are:

### Purpose

Indicates whether the implementation supports Flash Patch remap and, if it does, holds the target address for remap.

### Usage constraints

Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are **UNPREDICTABLE**.

If the Main Extension is not implemented then it is **IMPLEMENTATION DEFINED** whether this register is accessible only to the debugger and **RES0** for software. Otherwise the register is accessible to the debugger and software.

### Configurations

Present only if the FPB is implemented.

This register is **RES0** if the FPB is not implemented.

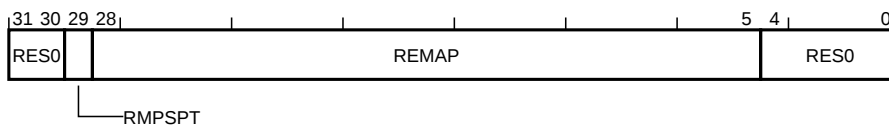
### Attributes

32-bit read-only register located at 0xE0002004.

This register is not banked between Security states.

## Field descriptions

The FP\_REMAP bit assignments are:



### Bits [31:30]

Reserved, **RES0**.

### RMPSPT, bit [29]

Remap supported. This field is RAZ. Remapping is not supported in Armv8-M.

### REMAP, bits [28:5]

Remap address.

Reserved, **RES0**.

### Bits [4:0]

Reserved, **RES0**.

### D1.2.103 HFSR, HardFault Status Register

The HFSR characteristics are:

#### Purpose

Shows the cause of any HardFaults.

#### Usage constraints

Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are **UNPREDICTABLE**.

#### Configurations

Present only if the Main Extension is implemented.

This register is **RES0** if the Main Extension is not implemented.

#### Attributes

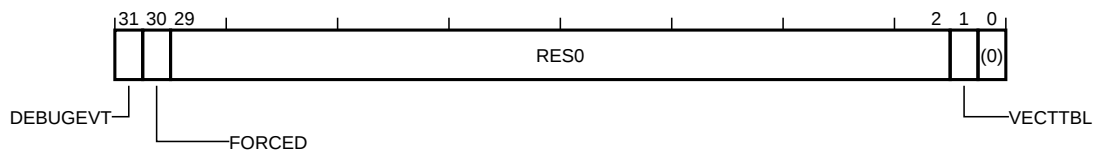
32-bit read/write-one-to-clear register located at 0xE000ED2C.

Secure software can access the Non-secure version of this register via HFSR\_NS located at 0xE002ED2C. The location 0xE002ED2C is **RES0** to software executing in Non-secure state and the debugger.

This register is not banked between Security states.

### Field descriptions

The HFSR bit assignments are:



#### DEBUGEVT, bit [31]

Debug event. Indicates when a debug event has occurred.

The possible values of this bit are:

**0**

No debug event has occurred.

**1**

Debug event has occurred. The Debug Fault Status Register has been updated.

The PE sets this bit to 1 only when Halting debug is disabled and a debug event occurs, except in the case where a BKPT instruction is escalated to HardFault. When AIRCR.BFHFNMIN is set to zero, the Non-secure view of this bit is RAZ/WI.

This bit resets to zero on a Warm reset.

#### FORCED, bit [30]

Forced. Indicates that a fault with configurable priority has been escalated to a HardFault exception, because it could not be made active, because of priority, or because it was disabled.

The possible values of this bit are:

**0**

No priority escalation has occurred.

**1**

Processor has escalated a configurable-priority exception to HardFault.

When AIRCR.BFHFNMINS is set to zero, the Non-secure view of this bit is RAZ/WI.

This bit resets to zero on a Warm reset.

**Bits [29:2]**

Reserved, RES0.

**VECTTBL, bit [1]**

Vector table. Indicates when a fault has occurred because of a vector table read error on exception processing.

The possible values of this bit are:

**0**

No vector table read fault has occurred.

**1**

Vector table read fault has occurred.

When AIRCR.BFHFNMINS is set to zero, the Non-secure view of this bit is RAZ/WI.

This bit resets to zero on a Warm reset.

**Bit [0]**

Reserved, RES0.

### D1.2.104 ICIALLU, Instruction Cache Invalidate All to PoU

The ICIALLU characteristics are:

#### Purpose

Invalidate all instruction caches to PoU.

#### Usage constraints

Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

#### Configurations

This register is always implemented.

#### Attributes

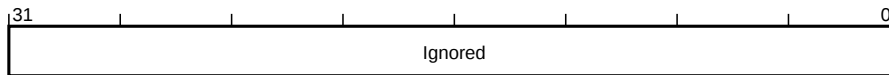
32-bit write-only register located at 0xE000EF50.

Secure software can access the Non-secure version of this register via ICIALLU\_NS located at 0xE002EF50. The location 0xE002EF50 is RES0 to software executing in Non-secure state and the debugger.

This register is not banked between Security states.

### Field descriptions

The ICIALLU bit assignments are:



#### Ignored, bits [31:0]

The value written to this field is ignored. Ignored.

### D1.2.105 ICIMVAU, Instruction Cache line Invalidate by Address to PoU

The ICIMVAU characteristics are:

#### Purpose

Invalidate instruction cache line by address to PoU.

#### Usage constraints

Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are [UNPREDICTABLE](#).

#### Configurations

This register is always implemented.

#### Attributes

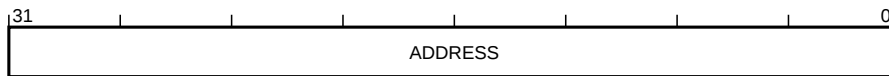
32-bit write-only register located at 0xE000EF58.

Secure software can access the Non-secure version of this register via ICIMVAU\_NS located at 0xE002EF58. The location 0xE002EF58 is [RES0](#) to software executing in Non-secure state and the debugger.

This register is not banked between Security states.

#### Field descriptions

The ICIMVAU bit assignments are:



#### ADDRESS, bits [31:0]

Address. Writing to this field initiates the maintenance operation for the address written.

### D1.2.106 ICSR, Interrupt Control and State Register

The ICSR characteristics are:

**Purpose**

Controls and provides status information for NMI, PendSV, SysTick and interrupts.

**Usage constraints**

Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

**Configurations**

This register is always implemented.

**Attributes**

32-bit read/write register located at 0xE00ED04.

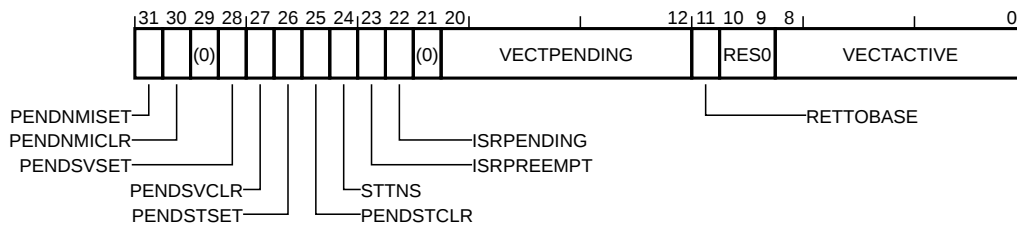
Secure software can access the Non-secure version of this register via ICSR\_NS located at 0xE002ED04. The location 0xE002ED04 is RES0 to software executing in Non-secure state and the debugger.

This register is banked between Security states on a bit by bit basis.

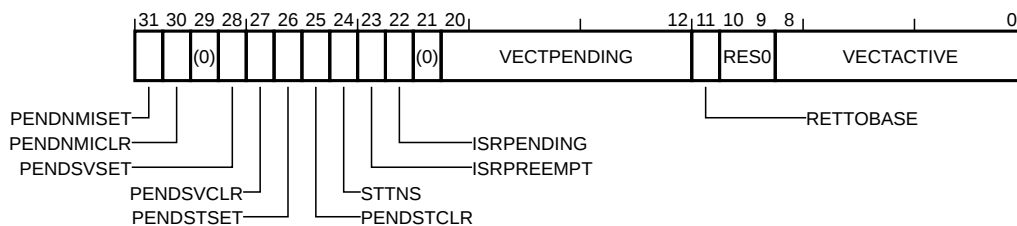
### Field descriptions

The ICSR bit assignments are:

On a read:



On a write:



**PENDNMISSET, bit [31], on a write**

Pend NMI set. Allows the NMI exception to be set as pending.

This bit is not banked between Security states.

The possible values of this bit are:

- 0**  
No effect.
- 1**  
Sets the NMI exception pending.

If both PENDNMISET and PENDNMICLR are written to one simultaneously, the pending state of the NMI exception becomes **UNKNOWN**.

This bit is write-one-to-set. Writes of zero are ignored.

If AIRCR.BFHFNMINS is zero this bit is RAZ/WI from Non-secure state.

**PENDNMISET, bit [31], on a read**

Pend NMI set. Indicates whether the NMI exception is pending.

This bit is not banked between Security states.

The possible values of this bit are:

**0**

NMI exception not pending.

**1**

NMI exception pending.

If AIRCR.BFHFNMINS is zero this bit is RAZ/WI from Non-secure state.

This bit resets to zero on a Warm reset.

**PENDNMICLR, bit [30]**

Pend NMI clear. Allows the NMI exception pending state to be cleared.

This bit is not banked between Security states.

The possible values of this bit are:

**0**

No effect.

**1**

Clear pending status.

This bit is write-only, and reads-as-zero.

This bit is write-one-to-clear. Writes of zero are ignored.

If AIRCR.BFHFNMINS is zero this bit is RAZ/WI from Non-secure state.

**Bit [29]**

Reserved, **RES0**.

**PENDSVSET, bit [28], on a write**

Pend PendSV set. Allows the PendSV exception for the selected Security state to be set as pending.

This bit is banked between Security states.

The possible values of this bit are:

**0**

No effect.

**1**

Sets the PendSV exception pending.

If both PENDSVSET and PENDSVCLR are written to one simultaneously, the pending state of the associated PendSV exception becomes **UNKNOWN**.

This bit is write-one-to-set. Writes of zero are ignored.

**PENDSVSET, bit [28], on a read**

Pend PendSV set. Indicates whether the PendSV for the selected Security state exception is pending.

This bit is banked between Security states.

The possible values of this bit are:

**0**

PendSV exception not pending.

**1**

PendSV exception pending.

This bit resets to zero on a Warm reset.

**PENDSVCLR, bit [27]**

Pend PendSV clear. Allows the PendSV exception pending state to be cleared for the selected Security state.

This bit is banked between Security states.

The possible values of this bit are:

**0**

No effect.

**1**

Clear pending status.

This bit is write-only, and reads-as-zero.

This bit is write-one-to-clear. Writes of zero are ignored.

**PENDSTSET, bit [26], on a write**

Pend SysTick set. Allows the SysTick for the selected Security state exception to be set as pending.

If two SysTick timers are implemented this bit is banked between Security states.

If less than two SysTick timers are implemented this bit is not banked between Security states.

The possible values of this bit are:

**0**

No effect.

**1**

Sets the SysTick exception for the selected Security state pending.

If only one SysTick timer is implemented and ICSR.STTNS is clear, this bit is RAZ/WI from Non-secure state.

If no SysTick timer is implemented this bit is [RES0](#).

**PENDSTSET, bit [26], on a read**

Pend SysTick set. Indicates whether the SysTick for the selected Security state exception is pending.

If two SysTick timers are implemented this bit is banked between Security states.

If less than two SysTick timers are implemented this bit is not banked between Security states.

The possible values of this bit are:

**0**

SysTick exception not pending.

**1**

SysTick exception pending.

If both PENDSTSET and PENDSTCLR are written to one simultaneously, the pending state of the associated SysTick exception becomes [UNKNOWN](#).

If only one SysTick timer is implemented and ICSR.STTNS is clear, this bit is RAZ/WI from Non-secure state.



If no SysTick timer is implemented this bit is [RES0](#).

This bit resets to zero on a Warm reset.

**PENDSTCLR, bit [25]**

Pend SysTick clear. Allows the SysTick exception pending state to be cleared for the selected Security state.

If two SysTick timers are implemented this bit is banked between Security states.

If less than two SysTick timers are implemented this bit is not banked between Security states.

The possible values of this bit are:

**0**

No effect.

**1**

Clear pending status.

This bit is write-only, and reads-as-zero.

This bit is write-one-to-clear. Writes of zero are ignored.

If only one SysTick timer is implemented and ICSR.STTNS is clear, this bit is RAZ/WI from Non-secure state.

If no SysTick timer is implemented this bit is [RES0](#).

**STTNS, bit [24]**

SysTick Targets Non-secure. Controls whether in a single SysTick implementation, the SysTick is Secure or Non-secure.

This bit is not banked between Security states.

The possible values of this bit are:

**0**

SysTick is Secure.

**1**

SysTick is Non-secure.

Behaves as RAZ/WI when either no SysTick or both SysTick timers are implemented. In a PE with the Main Extension and Security Extension this bit is [RES0](#). This bit is RAZ/WI when accessed from the Non-secure state.

This bit is RAZ/WI from Non-secure state.

This bit resets to zero on a Warm reset.

**ISRPREEMPT, bit [23]**

Interrupt preempt. Indicates whether a pending exception will be handled on exit from Debug state.

This bit is not banked between Security states.

The possible values of this bit are:

**0**

Will not handle.

**1**

Will handle a pending exception.

The value of this bit is [UNKNOWN](#) when not in Debug state.

This bit is read-only.

If neither Halting debug or the Main Extension are implemented, this bit is [RES0](#).

**ISR\_PENDING, bit [22]**

Interrupt pending. Indicates whether an external interrupt, generated by the NVIC, is pending.

This bit is not banked between Security states.

The possible values of this bit are:

**0**

No external interrupt pending.

**1**

External interrupt pending.

This bit is read-only.

If neither Halting debug or the Main Extension are implemented, this bit is [RES0](#).

**Note**

The value of DHCSR.C\_MASKINTS is ignored.

**Bit [21]**

Reserved, [RES0](#).

**VECT\_PENDING, bits [20:12]**

Vector pending. The exception number of the highest priority pending and enabled interrupt.

This field is not banked between Security states.

The possible values of this field are:

**Zero**

No pending and enabled exception.

**Non zero**

Exception number.

This field is read-only.

**Note**

If DHCSR.C\_MASKINTS is set, the PendSV, SysTick, and configurable external interrupts are masked and will not be shown as pending in VECT\_PENDING.

**RETTOBASE, bit [11]**

Return to base. In Handler mode, indicates whether there is more than one active exception.

This bit is not banked between Security states.

The possible values of this bit are:

**0**

There is more than one active exception.

**1**

There is only one active exception.

In Thread mode the value of this bit is [UNKNOWN](#).

This bit is read-only.

If the Main Extension is not implemented, this bit is [RES0](#).

**Bits [10:9]**

Reserved, [RES0](#).

**VECTACTIVE, bits [8:0]**

Vector active. The exception number of the current executing exception.

This field is not banked between Security states.

The possible values of this field are:

**Zero**

Thread mode.

**Non zero**

Exception number.

This value is the same as the IPSR Exception number. When the IPSR value has been set to 1 because of a function call to Non-secure state, this field is also set to 1.

This field is read-only.

If neither Halting debug or the Main Extension are implemented, this field is [RES0](#).

### D1.2.107 ICTR, Interrupt Controller Type Register

The ICTR characteristics are:

#### Purpose

Provides information about the interrupt controller.

#### Usage constraints

Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are **UNPREDICTABLE**.

#### Configurations

This register is always implemented.

#### Attributes

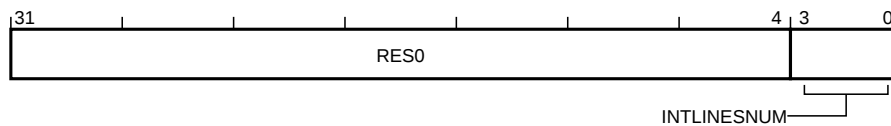
32-bit read-only register located at 0xE000E004.

Secure software can access the Non-secure version of this register via ICTR\_NS located at 0xE002E004. The location 0xE002E004 is **RES0** to software executing in Non-secure state and the debugger.

This register is not banked between Security states.

### Field descriptions

The ICTR bit assignments are:



#### Bits [31:4]

Reserved, **RES0**.

#### INTLINESNUM, bits [3:0]

Interrupt line set number. Indicates the number of the highest implemented register in each of the NVIC control register sets, or in the case of NVIC\_IPR $n$ , 4xINTLINESNUM.

This field reads as an **IMPLEMENTATION DEFINED** value.

### D1.2.108 ID\_AFR0, Auxiliary Feature Register 0

The ID\_AFR0 characteristics are:

#### Purpose

Provides information about the **IMPLEMENTATION DEFINED** features of the PE.

#### Usage constraints

Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are **UNPREDICTABLE**.

#### Configurations

Present only if the Main Extension is implemented.

This register is **RES0** if the Main Extension is not implemented.

#### Attributes

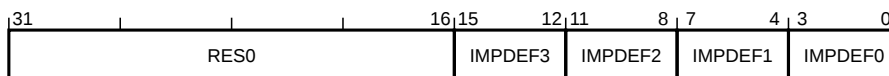
32-bit read-only register located at 0xE000ED4C.

Secure software can access the Non-secure version of this register via ID\_AFR0\_NS located at 0xE002ED4C. The location 0xE002ED4C is **RES0** to software executing in Non-secure state and the debugger.

This register is not banked between Security states.

### Field descriptions

The ID\_AFR0 bit assignments are:



#### Bits [31:16]

Reserved, **RES0**.

#### IMPDEF $m$ , bits [4 $m$ +3:4 $m$ ], for $m = 0$ to 3

**IMPLEMENTATION DEFINED**. **IMPLEMENTATION DEFINED** meaning.

This field reads as an **IMPLEMENTATION DEFINED** value.

## D1.2.109 ID\_DFR0, Debug Feature Register 0

The ID\_DFR0 characteristics are:

### Purpose

Provides top level information about the debug system.

### Usage constraints

Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

### Configurations

Present only if the Main Extension is implemented.

This register is RES0 if the Main Extension is not implemented.

### Attributes

32-bit read-only register located at 0xE000ED48.

Secure software can access the Non-secure version of this register via ID\_DFR0\_NS located at 0xE002ED48. The location 0xE002ED48 is RES0 to software executing in Non-secure state and the debugger.

This register is not banked between Security states.

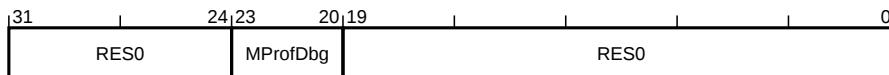
### Preface

If Halting debug is not implemented this register reads as 0x00000000.

If Halting debug is implemented this register reads as 0x00200000.

### Field descriptions

The ID\_DFR0 bit assignments are:



#### Bits [31:24]

Reserved, RES0.

#### MProfDbg, bits [23:20]

M-Profile debug. Indicates the supported M-Profile debug architecture.

The possible values of this field are:

**0b0000**

Halting debug is not implemented.

**0b0010**

Armv8-M Debug architecture.

All other values are reserved.

This field reads as an IMPLEMENTATION DEFINED value.

#### Bits [19:0]

Reserved, RES0.

## D1.2.110 ID\_ISAR0, Instruction Set Attribute Register 0

The ID\_ISAR0 characteristics are:

### Purpose

Provides information about the instruction set implemented by the PE.

### Usage constraints

Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are [UNPREDICTABLE](#).

### Configurations

Present only if the Main Extension is implemented.

This register is [RES0](#) if the Main Extension is not implemented.

### Attributes

32-bit read-only register located at 0xE000ED60.

Secure software can access the Non-secure version of this register via ID\_ISAR0\_NS located at 0xE002ED60. The location 0xE002ED60 is [RES0](#) to software executing in Non-secure state and the debugger.

This register is not banked between Security states.

### Preface

If coprocessors excluding the Floating-point Extension are not supported this register reads as 0x01101110.

If coprocessors excluding the Floating-point Extension are supported this register reads as 0x01141110.

### Field descriptions

The ID\_ISAR0 bit assignments are:

|      |        |       |        |           |          |          |      |   |
|------|--------|-------|--------|-----------|----------|----------|------|---|
| 31   | 28,27  | 24,23 | 20,19  | 16,15     | 12,11    | 8,7      | 4,3  | 0 |
| RES0 | Divide | Debug | Coproc | CmpBranch | BitField | BitCount | RES0 |   |

#### Bits [31:28]

Reserved, [RES0](#).

#### Divide, bits [27:24]

Divide. Indicates the supported Divide instructions.

The possible values of this field are:

**0b0001**

Supports SDIV and UDIV instructions.

All other values are reserved.

This field reads as 0b0001.

#### Debug, bits [23:20]

Debug. Indicates the implemented Debug instructions.

The possible values of this field are:

**0b0001**

Supports BKPT instruction.

All other values are reserved.

This field reads as 0b0001.

**Coproc, bits [19:16]**

Coprocessor. Indicates the supported coprocessor instructions.

The possible values of this field are:

**0b0000**

No coprocessor instructions support other than FPU.

**0b0100**

Coprocessor instructions supported.

All other values are reserved.

This field reads as an **IMPLEMENTATION DEFINED** value.

**CmpBranch, bits [15:12]**

Compare and branch. Indicates the supported combined Compare and Branch instructions.

The possible values of this field are:

**0b0001**

Supports CBNZ and CBZ instructions.

All other values are reserved.

This field reads as 0b0001.

**BitField, bits [11:8]**

Bit field. Indicates the supported bit field instructions.

The possible values of this field are:

**0b0001**

BFC, BFI, SBFX, and UBFX supported.

All other values are reserved.

This field reads as 0b0001.

**BitCount, bits [7:4]**

Bit count. Indicates the supported bit count instructions.

The possible values of this field are:

**0b0001**

CLZ supported.

All other values are reserved.

This field reads as 0b0001.

**Bits [3:0]**

Reserved, **RES0**.



### D1.2.111 ID\_ISAR1, Instruction Set Attribute Register 1

The ID\_ISAR1 characteristics are:

**Purpose**

Provides information about the instruction set implemented by the PE.

**Usage constraints**

Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

**Configurations**

Present only if the Main Extension is implemented.

This register is RES0 if the Main Extension is not implemented.

**Attributes**

32-bit read-only register located at 0xE000ED64.

Secure software can access the Non-secure version of this register via ID\_ISAR1\_NS located at 0xE002ED64. The location 0xE002ED64 is RES0 to software executing in Non-secure state and the debugger.

This register is not banked between Security states.

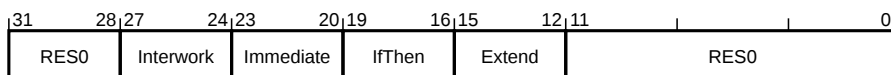
**Preface**

If the DSP Extension is not implemented, this register reads as 0x02211000.

If the DSP Extension is implemented, this register reads as 0x02212000.

**Field descriptions**

The ID\_ISAR1 bit assignments are:



**Bits [31:28]**

Reserved, RES0.

**Interwork, bits [27:24]**

Interworking. Indicates the implemented interworking instructions.

The possible values of this field are:

**0b0010**

BLX, BX, and loads to PC interwork.

All other values are reserved.

This field reads as 0b0010.

**Immediate, bits [23:20]**

Immediate. Indicates the implemented for data-processing instructions with long immediates.

The possible values of this field are:

**0b0010**

ADDW, MOVW, MOVT, and SUBW supported.

All other values are reserved.

This field reads as 0b0010.

**IfThen, bits [19:16]**

If-Then. Indicates the implemented If-Then instructions.

The possible values of this field are:

**0b0001**

IT instruction supported.

All other values are reserved.

This field reads as 0b0001.

**Extend, bits [15:12]**

Extend. Indicates the implemented Extend instructions.

The possible values of this field are:

**0b0001**

SXTB, SXTB, UXTB, and UXTH.

**0b0010**

Adds SXTB16, SXTAB, SXTAB16, SXTAH, UXTB16, UXTAB, UXTAB16, and UXTAH, DSP Extension only.

All other values are reserved.

This field reads as an **IMPLEMENTATION DEFINED** value.

**Bits [11:0]**

Reserved, **RES0**.

## D1.2.112 ID\_ISAR2, Instruction Set Attribute Register 2

The ID\_ISAR2 characteristics are:

### Purpose

Provides information about the instruction set implemented by the PE.

### Usage constraints

Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are **UNPREDICTABLE**.

### Configurations

Present only if the Main Extension is implemented.

This register is **RES0** if the Main Extension is not implemented.

### Attributes

32-bit read-only register located at 0xE000ED68.

Secure software can access the Non-secure version of this register via ID\_ISAR2\_NS located at 0xE002ED68. The location 0xE002ED68 is **RES0** to software executing in Non-secure state and the debugger.

This register is not banked between Security states.

### Preface

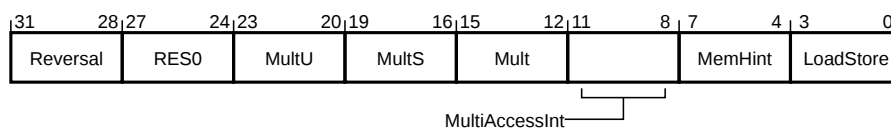
With bits [11:8] masked, if the DSP Extension is not implemented, this register reads as 0x20112032.

With bits[11:8] masked, if the DSP Extension is implemented, this register reads as 0x20232032.

The value of bits [11:8] is determined by whether the PE implements restartable or continuable multi-access instructions.

### Field descriptions

The ID\_ISAR2 bit assignments are:



#### Reversal, bits [31:28]

Reversal. Indicates the implemented Reversal instructions.

The possible values of this field are:

**0b0010**

REV, REV16, REVSH and RBIT instructions supported.

All other values are reserved.

This field reads as 0b0010.

#### Bits [27:24]

Reserved, **RES0**.

#### MultU, bits [23:20]

Multiply unsigned. Indicates the implemented advanced unsigned Multiply instructions.

The possible values of this field are:

**0b0001**

UMULL and UMLAL.

**0b0010**

Adds UMAAL, DSP Extension only.

All other values are reserved.

This field reads as an **IMPLEMENTATION DEFINED** value.

**MultS, bits [19:16]**

Multiply signed. Indicates the implemented advanced signed Multiply instructions.

The possible values of this field are:

**0b0001**

SMULL and SMLAL.

**0b0011**

Adds all saturating and DSP signed multiplies, DSP Extension only.

All other values are reserved.

This field reads as an **IMPLEMENTATION DEFINED** value.

**Mult, bits [15:12]**

Multiplies. Indicates the implemented additional Multiply instructions.

The possible values of this field are:

**0b0010**

MUL, MLA, and MLS.

All other values are reserved.

This field reads as 0b0010.

**MultiAccessInt, bits [11:8]**

Multi-access instructions. Indicates the support for interruptible multi-access instructions.

The possible values of this field are:

**0b0000**

No support. LDM and STM instructions are not interruptible.

**0b0001**

LDM and STM instructions are restartable.

**0b0010**

LDM and STM instructions are continuable.

All other values are reserved.

This field reads as an **IMPLEMENTATION DEFINED** value.

**MemHint, bits [7:4]**

Memory hints. Indicates the implemented Memory hint instructions.

The possible values of this field are:

**0b0011**

PLI and PLD instructions implemented.

All other values are reserved.

This field reads as 0b0011.

**LoadStore, bits [3:0]**

Load/store. Indicates the implemented additional load/store instructions.

The possible values of this field are:

**0b0010**

Supports load-acquire, store-release, and exclusive load and store instructions.

All other values are reserved.

This field reads as 0b0010.

### D1.2.113 ID\_ISAR3, Instruction Set Attribute Register 3

The ID\_ISAR3 characteristics are:

**Purpose**

Provides information about the instruction set implemented by the PE.

**Usage constraints**

Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

**Configurations**

Present only if the Main Extension is implemented.

This register is RES0 if the Main Extension is not implemented.

**Attributes**

32-bit read-only register located at 0xE000ED6C.

Secure software can access the Non-secure version of this register via ID\_ISAR3\_NS located at 0xE002ED6C. The location 0xE002ED6C is RES0 to software executing in Non-secure state and the debugger.

This register is not banked between Security states.

**Preface**

If the DSP Extension is not implemented, this register reads as 0x01111110.

If the DSP Extension is implemented, this register reads as 0x01111131.

**Field descriptions**

The ID\_ISAR3 bit assignments are:

|      |         |         |           |           |       |      |          |   |
|------|---------|---------|-----------|-----------|-------|------|----------|---|
| 31   | 28,27   | 24,23   | 20,19     | 16,15     | 12,11 | 8,7  | 4,3      | 0 |
| RES0 | TrueNOP | T32Copy | TabBranch | SynchPrim | SVC   | SIMD | Saturate |   |

**Bits [31:28]**

Reserved, RES0.

**TrueNOP, bits [27:24]**

True no-operation. Indicates the implemented true NOP instructions.

The possible values of this field are:

**0b0001**

NOP instruction and compatible hints implemented.

All other values are reserved.

This field reads as 0b0001.

**T32Copy, bits [23:20]**

T32 copy. Indicates the support for T32 non flag-setting MOV instructions.

The possible values of this field are:

**0b0001**

Encoding T1 of MOV (register) supports copying low register to low register.

All other values are reserved.

This field reads as 0b0001.

**TabBranch, bits [19:16]**

Table branch. Indicates the implemented Table Branch instructions.

The possible values of this field are:

**0b0001**

TBB and TBH implemented.

All other values are reserved.

This field reads as 0b0001.

**SynchPrim, bits [15:12]**

Synchronization primitives. Used in conjunction with ID\_ISAR4.SynchPrim\_frac to indicate the implemented synchronization primitive instructions.

The possible values of this field are:

**0b0001**

LDREX, STREX, LDREXB, STREXB, LDREXH, STREXH, and CLREX implemented.

All other values are reserved.

This field reads as 0b0001.

**SVC, bits [11:8]**

Supervisor Call. Indicates the implemented SVC instructions.

The possible values of this field are:

**0b0001**

SVC instruction implemented.

All other values are reserved.

This field reads as 0b0001.

**SIMD, bits [7:4]**

Single-instruction, multiple-data. Indicates the implemented SIMD instructions.

The possible values of this field are:

**0b0001**

SSAT, USAT, and Q-bit implemented.

**0b0011**

Adds all packed arithmetic and GE-bits, DSP Extension only.

All other values are reserved.

This field reads as an **IMPLEMENTATION DEFINED** value.

**Saturate, bits [3:0]**

Saturate. Indicates the implemented saturating instructions.

The possible values of this field are:

**0b0000**

None implemented.

**0b0001**

QADD, QDADD, QDSUB, QSUB, and Q-bit implemented, DSP Extension only.

*Chapter D1. Register Specification*

*D1.2. Alphabetical list of registers*

All other values are reserved.

This field reads as an **IMPLEMENTATION DEFINED** value.



### D1.2.114 ID\_ISAR4, Instruction Set Attribute Register 4

The ID\_ISAR4 characteristics are:

**Purpose**

Provides information about the instruction set implemented by the PE.

**Usage constraints**

Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

**Configurations**

Present only if the Main Extension is implemented.

This register is RES0 if the Main Extension is not implemented.

**Attributes**

32-bit read-only register located at 0xE000ED70.

Secure software can access the Non-secure version of this register via ID\_ISAR4\_NS located at 0xE002ED70. The location 0xE002ED70 is RES0 to software executing in Non-secure state and the debugger.

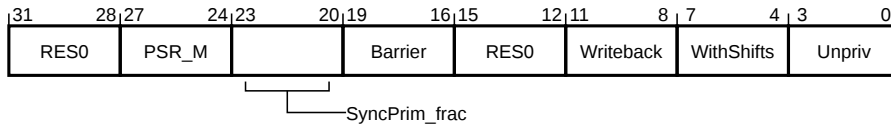
This register is not banked between Security states.

**Preface**

This register reads as 0x01310131.

**Field descriptions**

The ID\_ISAR4 bit assignments are:



**Bits [31:28]**

Reserved, RES0.

**PSR\_M, bits [27:24]**

Program Status Registers M. Indicates the implemented M profile instructions to modify the PSRs.

The possible values of this field are:

**0b0001**

M profile forms of CPS, MRS, and MSR implemented.

All other values are reserved.

This field reads as 0b0001.

**SyncPrim\_frac, bits [23:20]**

Synchronization primitives fractional. Used in conjunction with ID\_ISAR3.SynchPrim to indicate the implemented synchronization primitive instructions.

The possible values of this field are:

**0b0011**

LDREX, STREX, CLREX, LDREXB, LDREXH, STREXB, and STREXH implemented.

All other values are reserved.

This field reads as 0b0011.

**Barrier, bits [19:16]**

Barrier. Indicates the implemented Barrier instructions.

The possible values of this field are:

**0b0001**

CSDB, DMB, DSB, ISB, PSSBB and SSBB barrier instructions implemented.

All other values are reserved.

This field reads as 0b0001.

**Bits [15:12]**

Reserved, [RES0](#).

**Writeback, bits [11:8]**

Writeback. Indicates the support for writeback addressing modes.

The possible values of this field are:

**0b0001**

All writeback addressing modes supported.

All other values are reserved.

This field reads as 0b0001.

**WithShifts, bits [7:4]**

With shifts. Indicates the support for write-back addressing modes.

The possible values of this field are:

**0b0011**

Support for constant shifts on load/store and other instructions.

All other values are reserved.

This field reads as 0b0011.

**Unpriv, bits [3:0]**

Unprivileged. Indicates the implemented unprivileged instructions.

The possible values of this field are:

**0b0010**

LDRBT, LDRHT, LDRSBT, LDRSHT, LDRT, STRBT, STRHT, and STRT implemented.

All other values are reserved.

This field reads as 0b0010.

### D1.2.115 ID\_ISAR5, Instruction Set Attribute Register 5

The ID\_ISAR5 characteristics are:

#### Purpose

Provides information about the instruction set implemented by the PE.

#### Usage constraints

Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

#### Configurations

Present only if the Main Extension is implemented.

This register is RES0 if the Main Extension is not implemented.

#### Attributes

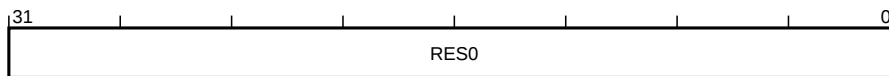
32-bit read-only register located at 0xE000ED74.

Secure software can access the Non-secure version of this register via ID\_ISAR5\_NS located at 0xE002ED74. The location 0xE002ED74 is RES0 to software executing in Non-secure state and the debugger.

This register is not banked between Security states.

### Field descriptions

The ID\_ISAR5 bit assignments are:



#### Bits [31:0]

Reserved, RES0.

## D1.2.116 ID\_MMFR0, Memory Model Feature Register 0

The ID\_MMFR0 characteristics are:

### Purpose

Provides information about the implemented memory model and memory management support.

### Usage constraints

Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are [UNPREDICTABLE](#).

### Configurations

Present only if the Main Extension is implemented.

This register is [RES0](#) if the Main Extension is not implemented.

### Attributes

32-bit read-only register located at 0xE000ED50.

Secure software can access the Non-secure version of this register via ID\_MMFR0\_NS located at 0xE002ED50. The location 0xE002ED50 is [RES0](#) to software executing in Non-secure state and the debugger.

This register is not banked between Security states.

## Field descriptions

The ID\_MMFR0 bit assignments are:

|      |    |    |    |        |    |     |    |          |   |      |   |      |   |
|------|----|----|----|--------|----|-----|----|----------|---|------|---|------|---|
| 31   | 24 | 23 | 20 | 19     | 16 | 15  | 12 | 11       | 8 | 7    | 4 | 3    | 0 |
| RES0 |    |    |    | AuxReg |    | TCM |    | ShareLvl |   | PMSA |   | RES0 |   |

### Bits [31:24]

Reserved, [RES0](#).

### AuxReg, bits [23:20]

Auxiliary Registers. Indicates support for Auxiliary Control Registers.

The possible values of this field are:

**0b0000**

No Auxiliary Control Registers.

**0b0001**

Auxiliary Control Registers supported.

All other values are reserved.

This field reads as an [IMPLEMENTATION DEFINED](#) value.

### TCM, bits [19:16]

Tightly Coupled Memories. Indicates support for Tightly Coupled Memories (TCMs).

The possible values of this field are:

**0b0000**

None supported.

**0b0001**

TCMs supported with [IMPLEMENTATION DEFINED](#) control.

All other values are reserved.

This field reads as an **IMPLEMENTATION DEFINED** value.

**ShareLvl, bits [15:12]**

Shareability Levels. Indicates the number of Shareability levels implemented.

The possible values of this field are:

**0b0000**

One level of Shareability implemented.

**0b0001**

Two levels of Shareability implemented.

All other values are reserved.

This field reads as an **IMPLEMENTATION DEFINED** value.

**OuterShr, bits [11:8]**

Outermost Shareability. Indicates the outermost Shareability domain implemented.

The possible values of this field are:

**0b0000**

Implemented as Non-cacheable.

**0b0001**

Implemented with hardware coherency support.

**0b1111**

Shareability ignored.

All other values are reserved.

This field reads as an **IMPLEMENTATION DEFINED** value.

**PMSA, bits [7:4]**

Protected memory system architecture. Indicates support for the protected memory system architecture (PMSA).

The possible values of this field are:

**0b0100**

Supports PMSAv8.

All other values are reserved.

This field reads as 0b0100.

**Bits [3:0]**

Reserved, **RES0**.

### D1.2.117 ID\_MMFR1, Memory Model Feature Register 1

The ID\_MMFR1 characteristics are:

#### Purpose

Provides information about the implemented memory model and memory management support.

#### Usage constraints

Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

#### Configurations

Present only if the Main Extension is implemented.

This register is RES0 if the Main Extension is not implemented.

#### Attributes

32-bit read-only register located at 0xE000ED54.

Secure software can access the Non-secure version of this register via ID\_MMFR1\_NS located at 0xE002ED54. The location 0xE002ED54 is RES0 to software executing in Non-secure state and the debugger.

This register is not banked between Security states.

### Field descriptions

The ID\_MMFR1 bit assignments are:



#### Bits [31:0]

Reserved, RES0.

## D1.2.118 ID\_MMFR2, Memory Model Feature Register 2

The ID\_MMFR2 characteristics are:

### Purpose

Provides information about the implemented memory model and memory management support.

### Usage constraints

Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are [UNPREDICTABLE](#).

### Configurations

Present only if the Main Extension is implemented.

This register is [RES0](#) if the Main Extension is not implemented.

### Attributes

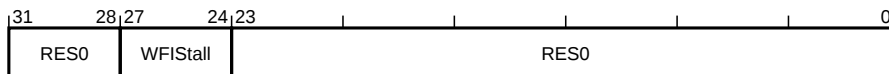
32-bit read-only register located at 0xE000ED58.

Secure software can access the Non-secure version of this register via ID\_MMFR2\_NS located at 0xE002ED58. The location 0xE002ED58 is [RES0](#) to software executing in Non-secure state and the debugger.

This register is not banked between Security states.

## Field descriptions

The ID\_MMFR2 bit assignments are:



### Bits [31:28]

Reserved, [RES0](#).

### WFISStall, bits [27:24]

WFI stall. Indicates the support for Wait For Interrupt (WFI) stalling.

The possible values of this field are:

**0b0000**

WFI never stalls.

**0b0001**

WFI has the ability to stall.

All other values are reserved.

This field reads as an [IMPLEMENTATION DEFINED](#) value.

### Bits [23:0]

Reserved, [RES0](#).

### D1.2.119 ID\_MMFR3, Memory Model Feature Register 3

The ID\_MMFR3 characteristics are:

#### Purpose

Provides information about the implemented memory model and memory management support.

#### Usage constraints

Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are [UNPREDICTABLE](#).

#### Configurations

Present only if the Main Extension is implemented.

This register is [RES0](#) if the Main Extension is not implemented.

#### Attributes

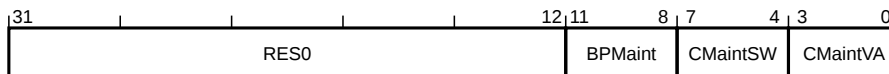
32-bit read-only register located at 0xE000ED5C.

Secure software can access the Non-secure version of this register via ID\_MMFR3\_NS located at 0xE002ED5C. The location 0xE002ED5C is [RES0](#) to software executing in Non-secure state and the debugger.

This register is not banked between Security states.

### Field descriptions

The ID\_MMFR3 bit assignments are:



#### Bits [31:12]

Reserved, [RES0](#).

#### BPMaint, bits [11:8]

Branch predictor maintenance. Indicates the supported branch predictor maintenance.

The possible values of this field are:

**0b0000**

None supported.

**0b0001**

Support for invalidate all of branch predictors.

All other values are reserved.

This field reads as an [IMPLEMENTATION DEFINED](#) value.

#### CMaintSW, bits [7:4]

Cache maintenance set/way. Indicates the supported cache maintenance operations by set/way.

The possible values of this field are:

**0b0000**

None supported.

**0b0001**

Maintenance by set/way operations supported.



All other values are reserved.

This field reads as an **IMPLEMENTATION DEFINED** value.

**CMaintVA, bits [3:0]**

Cache maintenance by address. Indicates the supported cache maintenance operations by address.

The possible values of this field are:

**0b0000**

None supported.

**0b0001**

Maintenance by address and instruction cache invalidate all supported.

All other values are reserved.

This field reads as an **IMPLEMENTATION DEFINED** value.





## D1.2.122 IPSR, Interrupt Program Status Register

The IPSR characteristics are:

### Purpose

Provides privileged access to the current exception number field.

### Usage constraints

Privileged access only. Unprivileged access is RAZ/WI, unless otherwise stated.

### Configurations

This register is always implemented.

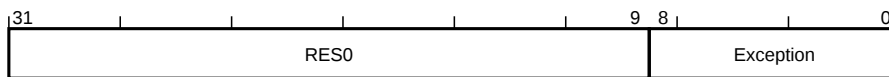
### Attributes

32-bit read/write special-purpose register.

This register is not banked between Security states.

## Field descriptions

The IPSR bit assignments are:



### Bits [31:9]

Reserved, **RES0**.

### Exception, bits [8:0]

Exception number. Holds the exception number of the currently-executing exception, or zero for Thread mode.

The possible values of this field are:

#### Zero

PE in Thread mode.

#### Non zero

PE in Handler mode in given exception number. On a function call from Secure state the value is set to 1 to ensure that the Non-secure state cannot determine which exception handler is executing.

This field resets to zero on a Warm reset.

### D1.2.123 ITM\_CIDR0, ITM Component Identification Register 0

The ITM\_CIDR0 characteristics are:

#### Purpose

Provides CoreSight discovery information for the ITM.

#### Usage constraints

If the Main Extension is implemented, both privileged and unprivileged accesses are permitted.

If the Main Extension is not implemented, unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are **UNPREDICTABLE**.

#### Configurations

Present only if CoreSight identification is implemented.

This register is **RES0** if CoreSight identification is not implemented.

Present only if the ITM is implemented.

This register is **RES0** if the ITM is not implemented.

If the Main Extension is not implemented then the ITM is not implemented.

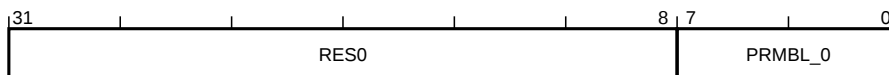
#### Attributes

32-bit read-only register located at 0xE0000FF0.

This register is not banked between Security states.

#### Field descriptions

The ITM\_CIDR0 bit assignments are:



#### Bits [31:8]

Reserved, **RES0**.

#### PRMBL\_0, bits [7:0]

CoreSight component identification preamble. See the *Arm® CoreSight™ Architecture Specification*.

This field reads as 0x0D.

### D1.2.124 ITM\_CIDR1, ITM Component Identification Register 1

The ITM\_CIDR1 characteristics are:

#### Purpose

Provides CoreSight discovery information for the ITM.

#### Usage constraints

If the Main Extension is implemented, both privileged and unprivileged accesses are permitted.

If the Main Extension is not implemented, unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are **UNPREDICTABLE**.

#### Configurations

Present only if CoreSight identification is implemented.

This register is **RES0** if CoreSight identification is not implemented.

Present only if the ITM is implemented.

This register is **RES0** if the ITM is not implemented.

If the Main Extension is not implemented then the ITM is not implemented.

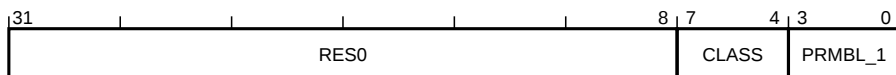
#### Attributes

32-bit read-only register located at 0xE0000FF4.

This register is not banked between Security states.

### Field descriptions

The ITM\_CIDR1 bit assignments are:



#### Bits [31:8]

Reserved, **RES0**.

#### CLASS, bits [7:4]

CoreSight component class. See the *Arm® CoreSight™ Architecture Specification*.

This field reads as 0x9.

#### PRMBL\_1, bits [3:0]

CoreSight component identification preamble. See the *Arm® CoreSight™ Architecture Specification*.

This field reads as 0x0.

## D1.2.125 ITM\_CIDR2, ITM Component Identification Register 2

The ITM\_CIDR2 characteristics are:

### Purpose

Provides CoreSight discovery information for the ITM.

### Usage constraints

If the Main Extension is implemented, both privileged and unprivileged accesses are permitted.

If the Main Extension is not implemented, unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are **UNPREDICTABLE**.

### Configurations

Present only if CoreSight identification is implemented.

This register is **RES0** if CoreSight identification is not implemented.

Present only if the ITM is implemented.

This register is **RES0** if the ITM is not implemented.

If the Main Extension is not implemented then the ITM is not implemented.

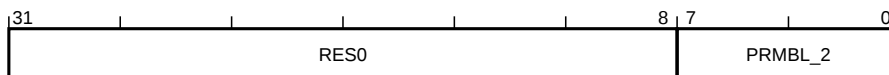
### Attributes

32-bit read-only register located at 0xE0000FF8.

This register is not banked between Security states.

## Field descriptions

The ITM\_CIDR2 bit assignments are:



### Bits [31:8]

Reserved, **RES0**.

### PRMBL\_2, bits [7:0]

CoreSight component identification preamble. See the *Arm® CoreSight™ Architecture Specification*.

This field reads as 0x05.

### D1.2.126 ITM\_CIDR3, ITM Component Identification Register 3

The ITM\_CIDR3 characteristics are:

#### Purpose

Provides CoreSight discovery information for the ITM.

#### Usage constraints

If the Main Extension is implemented, both privileged and unprivileged accesses are permitted.

If the Main Extension is not implemented, unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are **UNPREDICTABLE**.

#### Configurations

Present only if CoreSight identification is implemented.

This register is **RES0** if CoreSight identification is not implemented.

Present only if the ITM is implemented.

This register is **RES0** if the ITM is not implemented.

If the Main Extension is not implemented then the ITM is not implemented.

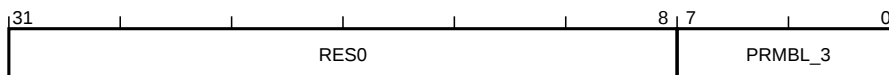
#### Attributes

32-bit read-only register located at 0xE0000FFC.

This register is not banked between Security states.

#### Field descriptions

The ITM\_CIDR3 bit assignments are:



#### Bits [31:8]

Reserved, **RES0**.

#### PRMBL\_3, bits [7:0]

CoreSight component identification preamble. See the *Arm® CoreSight™ Architecture Specification*.

This field reads as 0xB1.



## D1.2.127 ITM\_DEVARCH, ITM Device Architecture Register

The ITM\_DEVARCH characteristics are:

### Purpose

Provides CoreSight discovery information for the ITM.

### Usage constraints

If the Main Extension is implemented, both privileged and unprivileged accesses are permitted.

If the Main Extension is not implemented, unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are **UNPREDICTABLE**.

### Configurations

Present only if the Main Extension is implemented.

This register is **RES0** if the Main Extension is not implemented.

Present only if CoreSight identification is implemented.

This register is **RES0** if CoreSight identification is not implemented.

Present only if the ITM is implemented.

This register is **RES0** if the ITM is not implemented.

If the Main Extension is not implemented then the ITM is not implemented.

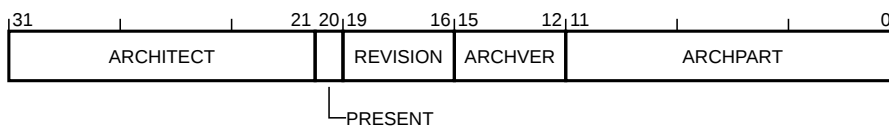
### Attributes

32-bit read-only register located at 0xE0000FBC.

This register is not banked between Security states.

## Field descriptions

The ITM\_DEVARCH bit assignments are:



### ARCHITECT, bits [31:21]

Architect. Defines the architect of the component. Bits [31:28] are the JEP106 continuation code (JEP106 bank ID, minus 1) and bits [27:21] are the JEP106 ID code.

The possible values of this field are:

#### 0x23B

JEP106 continuation code 0x4, ID code 0x3B. Arm Limited.

Other values are defined by the JEDEC JEP106 standard.

This field reads as 0x23B.

### PRESENT, bit [20]

DEVARCH Present. Defines that the DEVARCH register is present.

The possible values of this bit are:

#### 1

DEVARCH information present.

This bit reads as one.

**REVISION, bits [19:16]**

Revision. Defines the architecture revision of the component.

The possible values of this field are:

**0b0000**

ITM architecture v2.0.

This field reads as 0b0000.

**ARCHVER, bits [15:12]**

Architecture Version. Defines the architecture version of the component.

The possible values of this field are:

**0b0001**

ITM architecture v2.

ARCHVER and ARCHPART are also defined as a single field, ARCHID, so that ARCHVER is ARCHID[15:12].

This field reads as 0b0001.

**ARCHPART, bits [11:0]**

Architecture Part. Defines the architecture of the component.

The possible values of this field are:

**0xA01**

ITM architecture.

ARCHVER and ARCHPART are also defined as a single field, ARCHID, so that ARCHPART is ARCHID[11:0].

This field reads as 0xA01.



**0x0**

Miscellaneous.

**0x3**

Trace Source.

This field reads as an **IMPLEMENTATION DEFINED** value.

### D1.2.129 ITM\_LAR, ITM Software Lock Access Register

The ITM\_LAR characteristics are:

#### Purpose

Provides CoreSight Software Lock control for the ITM, see the *Arm® CoreSight™ Architecture Specification* for details.

#### Usage constraints

If the Main Extension is implemented, both privileged and unprivileged accesses are permitted, but unprivileged writes are ignored.

If the Main Extension is not implemented, unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are **UNPREDICTABLE**.

This register is RAZ/WI if accessed via the debugger.

#### Configurations

Present only if CoreSight identification is implemented.

This register is **RES0** if CoreSight identification is not implemented.

Present only if the ITM is implemented.

This register is **RES0** if the ITM is not implemented.

If the Main Extension is not implemented then the ITM is not implemented.

Present only if the optional Software Lock is implemented.

This register is RAZ/WI if the Software Lock is not implemented.

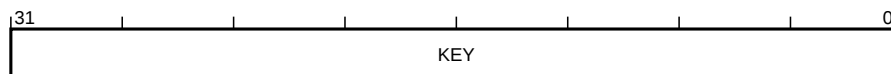
#### Attributes

32-bit write-only register located at 0xE0000FB0.

This register is not banked between Security states.

### Field descriptions

The ITM\_LAR bit assignments are:



#### KEY, bits [31:0]

Lock Access control.

Writing the key value 0xC5ACCE55 to this field unlocks the lock, enabling write accesses to the registers of this component through a memory-mapped interface.

Writing any other value to this register locks the lock, disabling write accesses to the registers of this component through a memory mapped interface.

### D1.2.130 ITM\_LSR, ITM Software Lock Status Register

The ITM\_LSR characteristics are:

**Purpose**

Provides CoreSight Software Lock status information for the ITM, see the *Arm® CoreSight™ Architecture Specification* for details.

**Usage constraints**

If the Main Extension is implemented, both privileged and unprivileged accesses are permitted.

If the Main Extension is not implemented, unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are **UNPREDICTABLE**.

This register is RAZ/WI if accessed via the debugger.

**Configurations**

Present only if CoreSight identification is implemented.

This register is **RES0** if CoreSight identification is not implemented.

Present only if the ITM is implemented.

This register is **RES0** if the ITM is not implemented.

If the Main Extension is not implemented then the ITM is not implemented.

Present only if the optional Software Lock is implemented.

This register is RAZ/WI if the Software Lock is not implemented.

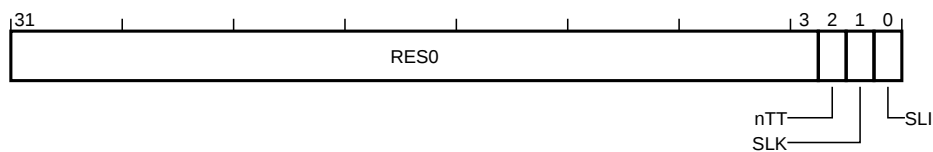
**Attributes**

32-bit read-only register located at 0xE0000FB4.

This register is not banked between Security states.

### Field descriptions

The ITM\_LSR bit assignments are:



**Bits [31:3]**

Reserved, **RES0**.

**nTT, bit [2]**

Not thirty-two bit. See the *Arm® CoreSight™ Architecture Specification*.

This bit reads as zero.

**SLK, bit [1]**

Software Lock status. See the *Arm® CoreSight™ Architecture Specification*.

The possible values of this bit are:

**0**

Lock clear. Software writes are permitted to the registers of this component.

**1**

Lock set. Software writes to the registers of this component are ignored, and reads have no side-effects.

For a debugger read of this register, or when the Software Lock is not implemented, this bit is **RES0**.

This bit resets to one on a Warm reset.

**SLI, bit [0]**

Software Lock implemented. See the *Arm® CoreSight™ Architecture Specification*.

The possible values of this bit are:

**0**

Software Lock not implemented or debugger access.

**1**

Software Lock is implemented and software access.

For a debugger read of this register, or when the Software Lock is not implemented, this bit is **RAZ**.

This bit reads as an **IMPLEMENTATION DEFINED** value.

### D1.2.131 ITM\_PIDR0, ITM Peripheral Identification Register 0

The ITM\_PIDR0 characteristics are:

#### Purpose

Provides CoreSight discovery information for the ITM.

#### Usage constraints

If the Main Extension is implemented, both privileged and unprivileged accesses are permitted.

If the Main Extension is not implemented, unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are **UNPREDICTABLE**.

#### Configurations

Present only if CoreSight identification is implemented.

This register is **RES0** if CoreSight identification is not implemented.

Present only if the ITM is implemented.

This register is **RES0** if the ITM is not implemented.

If the Main Extension is not implemented then the ITM is not implemented.

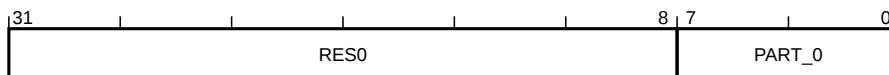
#### Attributes

32-bit read-only register located at 0xE0000FE0.

This register is not banked between Security states.

### Field descriptions

The ITM\_PIDR0 bit assignments are:



#### Bits [31:8]

Reserved, **RES0**.

#### PART\_0, bits [7:0]

Part number bits [7:0]. See the *Arm® CoreSight™ Architecture Specification*.

This field reads as an **IMPLEMENTATION DEFINED** value.



### D1.2.132 ITM\_PIDR1, ITM Peripheral Identification Register 1

The ITM\_PIDR1 characteristics are:

#### Purpose

Provides CoreSight discovery information for the ITM.

#### Usage constraints

If the Main Extension is implemented, both privileged and unprivileged accesses are permitted.

If the Main Extension is not implemented, unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are **UNPREDICTABLE**.

#### Configurations

Present only if CoreSight identification is implemented.

This register is **RES0** if CoreSight identification is not implemented.

Present only if the ITM is implemented.

This register is **RES0** if the ITM is not implemented.

If the Main Extension is not implemented then the ITM is not implemented.

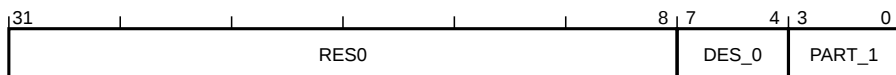
#### Attributes

32-bit read-only register located at 0xE0000FE4.

This register is not banked between Security states.

#### Field descriptions

The ITM\_PIDR1 bit assignments are:



#### Bits [31:8]

Reserved, **RES0**.

#### DES\_0, bits [7:4]

JEP106 identification code bits [3:0]. See the *Arm® CoreSight™ Architecture Specification*.

This field reads as an **IMPLEMENTATION DEFINED** value.

#### PART\_1, bits [3:0]

Part number bits [11:8]. See the *Arm® CoreSight™ Architecture Specification*.

This field reads as an **IMPLEMENTATION DEFINED** value.

### D1.2.133 ITM\_PIDR2, ITM Peripheral Identification Register 2

The ITM\_PIDR2 characteristics are:

#### Purpose

Provides CoreSight discovery information for the ITM.

#### Usage constraints

If the Main Extension is implemented, both privileged and unprivileged accesses are permitted.

If the Main Extension is not implemented, unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are **UNPREDICTABLE**.

#### Configurations

Present only if CoreSight identification is implemented.

This register is **RES0** if CoreSight identification is not implemented.

Present only if the ITM is implemented.

This register is **RES0** if the ITM is not implemented.

If the Main Extension is not implemented then the ITM is not implemented.

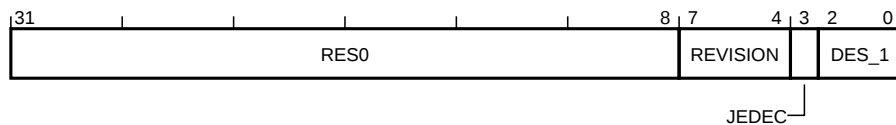
#### Attributes

32-bit read-only register located at 0xE0000FE8.

This register is not banked between Security states.

#### Field descriptions

The ITM\_PIDR2 bit assignments are:



#### Bits [31:8]

Reserved, **RES0**.

#### REVISION, bits [7:4]

Component revision. See the *Arm® CoreSight™ Architecture Specification*.

This field reads as an **IMPLEMENTATION DEFINED** value.

#### JEDEC, bit [3]

JEDEC assignee value is used. See the *Arm® CoreSight™ Architecture Specification*.

This bit reads as one.

#### DES\_1, bits [2:0]

JEP106 identification code bits [6:4]. See the *Arm® CoreSight™ Architecture Specification*.

This field reads as an **IMPLEMENTATION DEFINED** value.





### D1.2.136 ITM\_PIDR5, ITM Peripheral Identification Register 5

The ITM\_PIDR5 characteristics are:

#### Purpose

Provides CoreSight discovery information for the ITM.

#### Usage constraints

If the Main Extension is implemented, both privileged and unprivileged accesses are permitted.

If the Main Extension is not implemented, unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are **UNPREDICTABLE**.

#### Configurations

Present only if CoreSight identification is implemented.

This register is **RES0** if CoreSight identification is not implemented.

Present only if the ITM is implemented.

This register is **RES0** if the ITM is not implemented.

If the Main Extension is not implemented then the ITM is not implemented.

#### Attributes

32-bit read-only register located at 0xE0000FD4.

This register is not banked between Security states.

#### Field descriptions

The ITM\_PIDR5 bit assignments are:



#### Bits [31:0]

Reserved, **RES0**.

### D1.2.137 ITM\_PIDR6, ITM Peripheral Identification Register 6

The ITM\_PIDR6 characteristics are:

#### Purpose

Provides CoreSight discovery information for the ITM.

#### Usage constraints

If the Main Extension is implemented, both privileged and unprivileged accesses are permitted.

If the Main Extension is not implemented, unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are **UNPREDICTABLE**.

#### Configurations

Present only if CoreSight identification is implemented.

This register is **RES0** if CoreSight identification is not implemented.

Present only if the ITM is implemented.

This register is **RES0** if the ITM is not implemented.

If the Main Extension is not implemented then the ITM is not implemented.

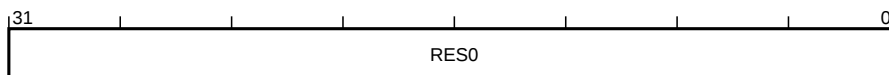
#### Attributes

32-bit read-only register located at 0xE0000FD8.

This register is not banked between Security states.

#### Field descriptions

The ITM\_PIDR6 bit assignments are:



#### Bits [31:0]

Reserved, **RES0**.

### D1.2.138 ITM\_PIDR7, ITM Peripheral Identification Register 7

The ITM\_PIDR7 characteristics are:

#### Purpose

Provides CoreSight discovery information for the ITM.

#### Usage constraints

If the Main Extension is implemented, both privileged and unprivileged accesses are permitted.

If the Main Extension is not implemented, unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are **UNPREDICTABLE**.

#### Configurations

Present only if CoreSight identification is implemented.

This register is **RES0** if CoreSight identification is not implemented.

Present only if the ITM is implemented.

This register is **RES0** if the ITM is not implemented.

If the Main Extension is not implemented then the ITM is not implemented.

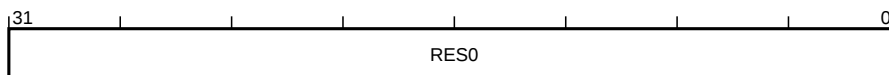
#### Attributes

32-bit read-only register located at 0xE000FDC.

This register is not banked between Security states.

#### Field descriptions

The ITM\_PIDR7 bit assignments are:



#### Bits [31:0]

Reserved, **RES0**.

### D1.2.139 ITM\_STIMn, ITM Stimulus Port Register, n = 0 - 255

The ITM\_STIM{0..255} characteristics are:

**Purpose**

Provides the interface for generating Instrumentation packets.

**Usage constraints**

If the Main Extension is implemented, both privileged and unprivileged accesses are permitted, but unprivileged writes are ignored if ITM\_TPR.PRIVMASK[n DIV 8] is set to one.

If the Main Extension is not implemented, unprivileged accesses generate a fault.

All writes are ignored if ITM\_TCR.ITMENA == 0 or ITM\_TER{n DIV 32}.STIMENA[n MOD 32] == 0.

This register is word, halfword, and byte accessible.

Accesses that are not word aligned are [UNPREDICTABLE](#).

**Configurations**

Present only if the ITM is implemented.

This register is [RES0](#) if the ITM is not implemented.

If the Main Extension is not implemented then the ITM is not implemented.

**Attributes**

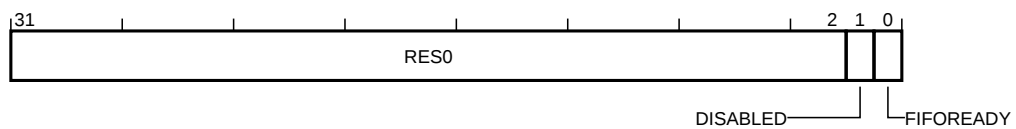
32-bit read/write register located at 0xE0000000 + 4n.

This register is not banked between Security states.

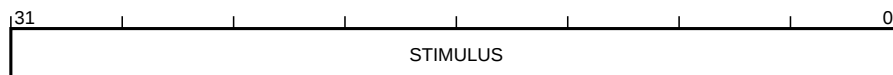
### Field descriptions

The ITM\_STIM{0..255} bit assignments are:

On a read:



On a write:



**STIMULUS, bits [31:0], on a write**

Stimulus data. Data to write to the stimulus port output buffer, for forwarding as an Instrumentation packet. The size of write access determines the type of Instrumentation packet generated.

**Bits [31:2], on a read**

Reserved, [RES0](#).

**DISABLED, bit [1], on a read**

Disabled. Indicates whether the stimulus port is enabled or disabled.

The possible values of this bit are:

- 0**  
Stimulus port and ITM are enabled.



**1**

Stimulus port or ITM is disabled.

**FIFOREADY, bit [0], on a read**

FIFO ready. Indicates whether the stimulus port can accept data.

The possible values of this bit are:

**0**

Stimulus port cannot accept data.

**1**

Stimulus port can accept at least one word.

### D1.2.140 ITM\_TCR, ITM Trace Control Register

The ITM\_TCR characteristics are:

**Purpose**

Configures and controls transfers through the ITM interface.

**Usage constraints**

If the Main Extension is implemented, both privileged and unprivileged accesses are permitted, but unprivileged writes are ignored.

If the Main Extension is not implemented, unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are **UNPREDICTABLE**.

**Configurations**

Present only if the ITM is implemented.

This register is **RES0** if the ITM is not implemented.

If the Main Extension is not implemented then the ITM is not implemented.

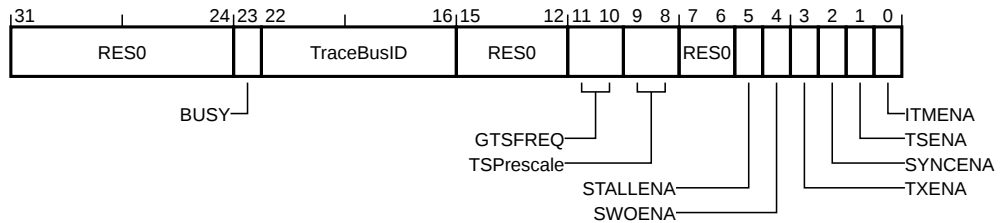
**Attributes**

32-bit read/write register located at 0xE0000E80.

This register is not banked between Security states.

### Field descriptions

The ITM\_TCR bit assignments are:



**Bits [31:24]**

Reserved, **RES0**.

**BUSY, bit [23]**

ITM busy. Indicates whether the ITM is currently processing events.

The possible values of this bit are:

**0**  
ITM is not processing any events.

**1**  
Events present and being drained.

Events means the ITM is generating or processing any of:

- Packets generated by the ITM from writes to Stimulus Ports.
- Other packets generated by the ITM itself.
- Packets generated by the DWT.

This bit is read-only.

#### **TraceBusID, bits [22:16]**

Trace bus identity. Identifier for multi-source trace stream formatting. If multi-source trace is in use, the debugger must write a unique nonzero trace ID value to this field.

The possible values of this field are:

**0x00**

Multi-source trace not in use.

**0x01-0x6F**

Unique trace ID value to be used for ITM trace packets.

All other values are reserved. If the ITM is the only trace source in the system, this field might be RAZ.

This field resets to an **UNKNOWN** value on a Cold reset.

#### **Bits [15:12]**

Reserved, **RES0**.

#### **GTSFREQ, bits [11:10]**

Global timestamp frequency. Defines how often the ITM generates a global timestamp, based on the global timestamp clock frequency, or disables generation of global timestamps.

The possible values of this field are:

**0b00**

Disable generation of Global Timestamp packets.

**0b01**

Generate timestamp request whenever the ITM detects a change in global timestamp counter bits [ $N-1:7$ ]. This is approximately every 128 cycles.

**0b10**

Generate timestamp request whenever the ITM detects a change in global timestamp counter bits [ $N-1:13$ ]. This is approximately every 8192 cycles.

**0b11**

Generate a timestamp after every packet, if the output FIFO is empty.

$N$  is the size of the global timestamp counter.

If the implementation does not support global timestamping then these bits are reserved, RAZ/WI.

This field resets to zero on a Cold reset.

#### **TSPrescale, bits [9:8]**

Timestamp prescale. Local timestamp prescaler, used with the trace packet reference clock.

The possible values of this field are:

**0b00**

No prescaling.

**0b01**

Divide by 4.

**0b10**

Divide by 16.

**0b11**

Divide by 64.

If the processor does not implement the timestamp prescaler then these bits are reserved, RAZ/WI.

This field resets to zero on a Cold reset.

**Bits [7:6]**

Reserved, [RES0](#).

**STALLENA, bit [5]**

Stall enable. Stall the PE to guarantee delivery of Data Trace packets.

The possible values of this bit are:

**0**

Drop Hardware Source packets and generate an Overflow packet if the ITM output is stalled.

**1**

Stall the PE to guarantee delivery of Data Trace packets.

If stalling is not implemented, this bit is RAZ/WI.

**SWOENA, bit [4]**

SWO enable. Enables asynchronous clocking of the timestamp counter.

The possible values of this bit are:

**0**

Timestamp counter uses the processor system clock.

**1**

Timestamp counter uses asynchronous clock from the TPIU interface. The timestamp counter is held in reset while the output line is idle.

Which clocking modes are implemented is [IMPLEMENTATION DEFINED](#). If the implementation does not support both modes this bit is either RAZ or RAO, to indicate the implemented mode.

This bit resets to an [UNKNOWN](#) value on a Cold reset.

**TXENA, bit [3]**

Transmit enable. Enables forwarding of hardware event packet from the DWT unit to the ITM for output to the TPIU.

The possible values of this bit are:

**0**

Disabled.

**1**

Enabled.

It is [IMPLEMENTATION DEFINED](#) whether the DWT discards packets that it cannot forward to the ITM.

This bit resets to zero on a Cold reset.

**Note**

If a debugger changes this bit from 0 to 1, the DWT might forward a hardware event packet that it has previously generated.

**SYNCENA, bit [2]**

Synchronization enable. Enables Synchronization packet transmission for a synchronous TPIU.

The possible values of this bit are:

**0**

Disabled.

**1**

Enabled.

This bit resets to zero on a Cold reset.

**Note**

If a debugger sets this bit to 1 it must also configure DWT\_CTRL.SYNCTAP for the correct synchronization speed.

**TSENA, bit [1]**

Timestamp enable. Enables Local timestamp generation.

The possible values of this bit are:

**0**  
Disabled.

**1**  
Enabled.

This bit resets to zero on a Cold reset.

**ITMENA, bit [0]**

ITM enable. Enables the ITM.

The possible values of this bit are:

**0**  
Disabled.

**1**  
Enabled.

This is the master enable for the ITM unit. A debugger must set this bit to 1 to permit writes to all Stimulus Port registers.

This bit resets to zero on a Cold reset.

### D1.2.141 ITM\_TERN, ITM Trace Enable Register, n = 0 - 7

The ITM\_TER{0..7} characteristics are:

#### Purpose

Provide an individual enable bit for each ITM\_STIM register.

#### Usage constraints

If the Main Extension is implemented, both privileged and unprivileged accesses are permitted.

If the Main Extension is not implemented, unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

#### Configurations

Present only if the ITM is implemented.

This register is RES0 if the ITM is not implemented.

If the Main Extension is not implemented then the ITM is not implemented.

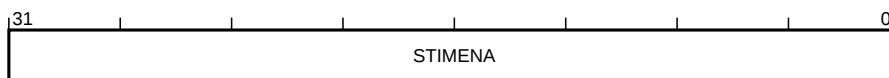
#### Attributes

32-bit read/write register located at  $0xE0000E00 + 4n$ .

This register is not banked between Security states.

### Field descriptions

The ITM\_TER{0..7} bit assignments are:



#### STIMENA, bits [31:0]

Stimulus enable. For STIMENA[m] in ITM\_TERN, controls whether stimulus port ITM\_STIM<32n+m> is enabled.

The possible values of each bit are:

**0**

Stimulus port (32n + m) disabled.

**1**

Stimulus port (32n + m) enabled.

Bits corresponding to unimplemented stimulus ports are RAZ/WI. Unprivileged writes to ITM\_TERN do not update STIMENA[m] if ITM\_TPR.PRIVMASK[(32n+m) DIV 8] is set to 1.

This field resets to zero on a Cold reset.

### D1.2.142 ITM\_TPR, ITM Trace Privilege Register

The ITM\_TPR characteristics are:

#### Purpose

Controls which stimulus ports can be accessed by unprivileged code.

#### Usage constraints

If the Main Extension is implemented, both privileged and unprivileged accesses are permitted, but unprivileged writes are ignored.

If the Main Extension is not implemented, unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are **UNPREDICTABLE**.

#### Configurations

Present only if the ITM is implemented.

This register is **RES0** if the ITM is not implemented.

If the Main Extension is not implemented then the ITM is not implemented.

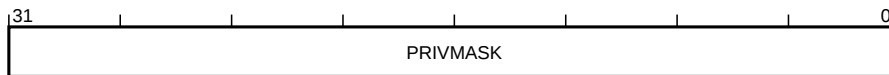
#### Attributes

32-bit read/write register located at 0xE0000E40.

This register is not banked between Security states.

### Field descriptions

The ITM\_TPR bit assignments are:



#### PRIVMASK, bits [31:0]

Privilege mask. For PRIVMASK[*m*], defines the access permissions of stimulus ports ITM\_STIM<8*m*> to ITM\_STIM<8*m*+7> inclusive.

The possible values of each bit are:

**0**

Unprivileged access permitted.

**1**

Privileged access only.

Bits corresponding to unimplemented stimulus ports are RAZ/WI.

This field resets to zero on a Cold reset.

### D1.2.143 LR, Link Register

The LR characteristics are:

**Purpose**

Exception and procedure call link register.

**Usage constraints**

Privileged and unprivileged access permitted.

**Configurations**

This register is always implemented.

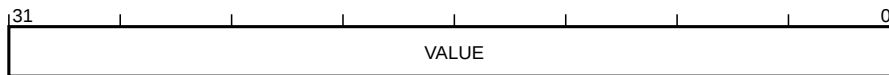
**Attributes**

32-bit read/write special-purpose register.

This register is not banked between Security states.

### Field descriptions

The LR bit assignments are:



**VALUE, bits [31:0]**

Link register. 32-bit link register updated to hold a return address, FNC\_RETURN or EXC\_RETURN on a function call or exception entry. LR can be used as a general-purpose register.

This field resets to an UNKNOWN value on Warm reset when the Main Extension is not implemented.

This field resets to 0xFFFFFFFF on a Warm reset if the Main Extension is implemented.



### D1.2.144 MAIR\_ATTR, Memory Attribute Indirection Register Attributes

The MAIR\_ATTR characteristics are:

**Purpose**

Defines the memory attribute encoding for use in the MPU\_MAIR0 and MPU\_MAIR1.

**Usage constraints**

None.

**Configurations**

All.

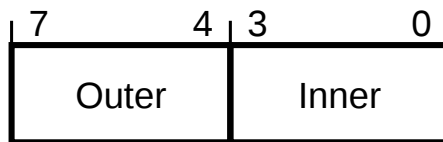
**Attributes**

8-bit payload.

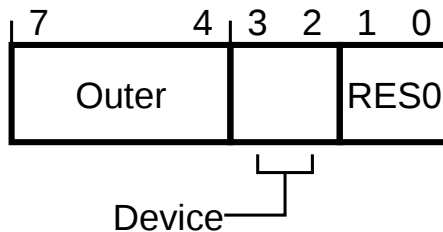
#### Field descriptions

The MAIR\_ATTR bit assignments are:

When Outer != 0b0000:



When Outer == 0b0000:



**Outer, bits [7:4]**

Outer attributes. Specifies the Outer memory attributes.

The possible values of this field are:

**0b0000**

Device memory.

**0b00RW**

Normal memory, Outer Write-Through transient (RW!=0b00).

**0b0100**

Normal memory, Outer Non-cacheable.

**0b01RW**

Normal memory, Outer Write-Back Transient (RW!=0b00).

**0b10RW**

Normal memory, Outer Write-Through Non-transient.

**0b11RW**

Normal memory, Outer Write-Back Non-transient.

R and W specify the outer read and write allocation policy: 0 = do not allocate, 1 = allocate.

**Device, bits [3:2], when Outer == 0b0000**

Device attributes. Specifies the memory attributes for Device.

The possible values of this field are:

**0b00**

Device-nGnRnE.

**0b01**

Device-nGnRE.

**0b10**

Device-nGRE.

**0b11**

Device-GRE.

**Bits [1:0 , when Outer == 0b0000]**

Reserved, [RES0](#).

**Inner, bits [3:0], when Outer != 0b0000**

Inner attributes. Specifies the Inner memory attributes.

The possible values of this field are:

**0b0000**

[UNPREDICTABLE](#).

**0b00RW**

Normal memory, Inner Write-Through Transient (RW!=0b00).

**0b0100**

Normal memory, Inner Non-cacheable.

**0b01RW**

Normal memory, Inner Write-Back Transient (RW!=0b00).

**0b10RW**

Normal memory, Inner Write-Through Non-transient.

**0b11RW**

Normal memory, Inner Write-Back Non-transient.

R and W specify the inner read and write allocation policy: 0 = do not allocate, 1 = allocate.

### D1.2.145 MMFAR, MemManage Fault Address Register

The MMFAR characteristics are:

#### Purpose

Shows the address of the memory location that caused an MPU fault.

#### Usage constraints

Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are **UNPREDICTABLE**.

#### Configurations

Present only if the Main Extension is implemented.

This register is **RES0** if the Main Extension is not implemented.

#### Attributes

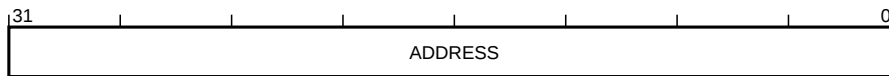
32-bit read/write register located at 0xE000ED34.

Secure software can access the Non-secure version of this register via MMFAR\_NS located at 0xE002ED34. The location 0xE002ED34 is **RES0** to software executing in Non-secure state and the debugger.

This register is banked between Security states.

### Field descriptions

The MMFAR bit assignments are:



#### ADDRESS, bits [31:0]

Data address for an MemManage fault. This register is updated with the address of a location that produced a MemManage fault. The MMFSR shows the cause of the fault, and whether this field is valid. This field is valid only when MMFSR.MMARVALID is set, otherwise it is **UNKNOWN**.

In implementations without unique BFAR and MMFAR registers, the value of this register is **UNKNOWN** if BFSR.BFARVALID is set.

This field resets to an **UNKNOWN** value on a Warm reset.

### D1.2.146 MMFSR, MemManage Fault Status Register

The MMFSR characteristics are:

**Purpose**

Shows the status of MPU faults.

**Usage constraints**

Privileged access permitted only. Unprivileged accesses generate a fault.

If the Main Extension is implemented, this register is word, halfword, and byte accessible.

If the Main Extension is not implemented, this register is word accessible only, halfword and byte accesses are **UNPREDICTABLE**.

**Configurations**

Present only if the Main Extension is implemented.

This register is **RES0** if the Main Extension is not implemented.

**Attributes**

8-bit read/write-one-to-clear register located at 0xE000ED28.

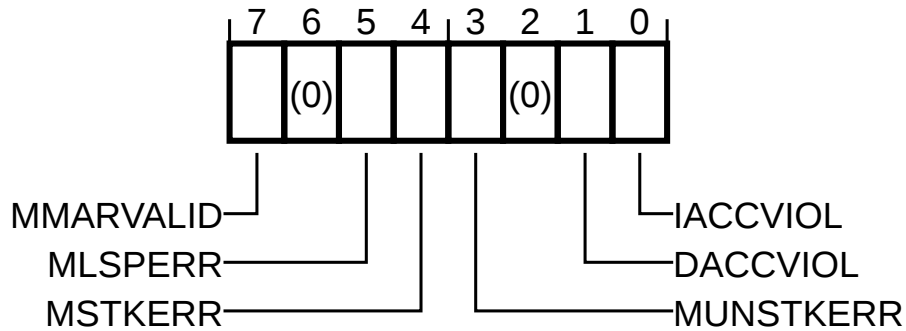
Secure software can access the Non-secure version of this register via MMFSR\_NS located at 0xE002ED28. The location 0xE002ED28 is **RES0** to software executing in Non-secure state and the debugger.

This register is banked between Security states.

This register is part of CFSR.

### Field descriptions

The MMFSR bit assignments are:



**MMARVALID, bit [7]**

MMFAR valid flag. Indicates validity of the MMFAR register.

The possible values of this bit are:

**0**  
MMFAR content not valid.

**1**  
MMFAR content valid.

This bit resets to zero on a Warm reset.

**Bit [6]**

Reserved, **RES0**.

**MLSPERR, bit [5]**

MemManage lazy state preservation error flag. Records whether a MemManage fault occurred during FP lazy state preservation.

The possible values of this bit are:

**0**

No MemManage occurred.

**1**

MemManage occurred.

This bit resets to zero on a Warm reset.

**MSTKERR, bit [4]**

MemManage stacking error flag. Records whether a derived MemManage fault occurred during exception entry stacking.

The possible values of this bit are:

**0**

No derived MemManage occurred.

**1**

Derived MemManage occurred during exception entry.

This bit resets to zero on a Warm reset.

**MUNSTKERR, bit [3]**

MemManage unstacking error flag. Records whether a derived MemManage fault occurred during exception return unstacking.

The possible values of this bit are:

**0**

No derived MemManage fault occurred.

**1**

Derived MemManage fault occurred during exception return.

This bit resets to zero on a Warm reset.

**Bit [2]**

Reserved, [RES0](#).

**DACCVIOL, bit [1]**

Data access violation flag. Records whether a data access violation has occurred.

The possible values of this bit are:

**0**

No MemManage fault on data access has occurred.

**1**

MemManage fault on data access has occurred.

A DACCVIOL will be accompanied by an MMFAR update.

This bit resets to zero on a Warm reset.

**IACCVIOL, bit [0]**

Instruction access violation. Records whether an instruction related memory access violation has occurred.

The possible values of this bit are:

**0**

No MemManage fault on instruction access has occurred.

**1**

MemManage fault on instruction access has occurred.

An IACCVIOL is only recorded if a faulted instruction is executed.

This bit resets to zero on a Warm reset.

## D1.2.147 MPU\_CTRL, MPU Control Register

The MPU\_CTRL characteristics are:

### Purpose

Enables the MPU and, when the MPU is enabled, controls whether the default memory map is enabled as a background region for privileged accesses, and whether the MPU is enabled for HardFaults, NMIs, and exception handlers when FAULTMASK is set to 1.

### Usage constraints

Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are **UNPREDICTABLE**.

### Configurations

This register is always implemented.

### Attributes

32-bit read/write register located at 0xE000ED94.

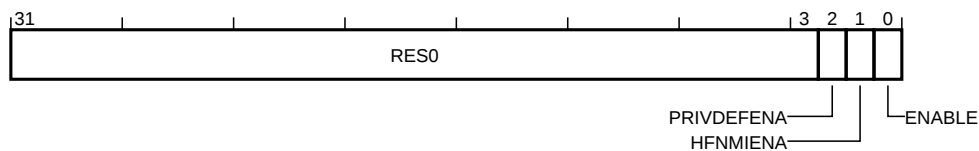
To allow lock down of this register it is **IMPLEMENTATION DEFINED** whether this register is writable.

Secure software can access the Non-secure version of this register via MPU\_CTRL\_NS located at 0xE002ED94. The location 0xE002ED94 is **RES0** to software executing in Non-secure state and the debugger.

This register is banked between Security states.

## Field descriptions

The MPU\_CTRL bit assignments are:



### Bits [31:3]

Reserved, **RES0**.

### PRIVDEFENA, bit [2]

Privileged default enable. Controls whether the default memory map is enabled for privileged software.

The possible values of this bit are:

**0**

Use of default memory map disabled.

**1**

Use of default memory map enabled for privilege code.

When the ENABLE bit is set to 0, the PE ignores the PRIVDEFENA bit. If no regions are enabled and the PRIVDEFENA and ENABLE bits are set to 1, only privileged code can execute from the system address map. If no MPU regions are implemented this bit is **RES0**.

This bit resets to zero on a Warm reset.

### HFNMIENA, bit [1]

HardFault, NMI enable. Controls whether handlers executing with a requested execution priority of less than 0 access memory with the MPU enabled or disabled. This applies to HardFaults and NMIs when FAULTMASK is set to 1.

The possible values of this bit are:

**0**

MPU disabled for these handlers.

**1**

MPU enabled for these handlers.

If HFNMIENA is set to 1 when ENABLE is set to 0, behavior is **UNPREDICTABLE**. If no MPU regions are implemented this bit is **RES0**.

This bit resets to zero on a Warm reset.

**ENABLE, bit [0]**

Enable. Enables the MPU.

The possible values of this bit are:

**0**

The MPU is disabled.

**1**

The MPU is enabled.

Disabling the MPU, by setting the ENABLE bit to 0, means that privileged and unprivileged accesses use the default memory map. If no MPU regions are implemented this bit is **RES0**.

This bit resets to zero on a Warm reset.



### D1.2.148 MPU\_MAIR0, MPU Memory Attribute Indirection Register 0

The MPU\_MAIR0 characteristics are:

#### Purpose

Along with MPU\_MAIR1, provides the memory attribute encodings corresponding to the AttrIdx values.

#### Usage constraints

Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

#### Configurations

This register is always implemented.

#### Attributes

32-bit read/write register located at 0xE000EDC0.

To allow lock down of this register it is IMPLEMENTATION DEFINED whether this register is writable.

Secure software can access the Non-secure version of this register via MPU\_MAIR0\_NS located at 0xE002EDC0. The location 0xE002EDC0 is RES0 to software executing in Non-secure state and the debugger.

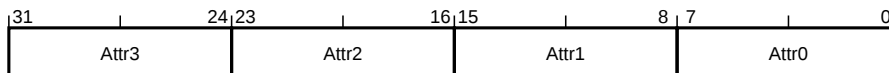
This register is banked between Security states.

#### Preface

This register is RES0 if no MPU regions are implemented in the corresponding Security state.

#### Field descriptions

The MPU\_MAIR0 bit assignments are:



#### Attr $m$ , bits [8 $m$ +7:8 $m$ ], for $m = 0$ to 3

Attribute  $m$ . Memory attribute encoding for MPU regions with an AttrIdx of  $m$ .

The possible values of this field are:

#### All

See MAIR\_ATTR for encoding.

This field resets to an Architecturally UNKNOWN value on a Warm reset.

### D1.2.149 MPU\_MAIR1, MPU Memory Attribute Indirection Register 1

The MPU\_MAIR1 characteristics are:

#### Purpose

Along with MPU\_MAIR0, provides the memory attribute encodings corresponding to the AttrIdx values.

#### Usage constraints

Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

#### Configurations

This register is always implemented.

#### Attributes

32-bit read/write register located at 0xE000EDC4.

To allow lock down of this register it is IMPLEMENTATION DEFINED whether this register is writable.

Secure software can access the Non-secure version of this register via MPU\_MAIR1\_NS located at 0xE002EDC4. The location 0xE002EDC4 is RES0 to software executing in Non-secure state and the debugger.

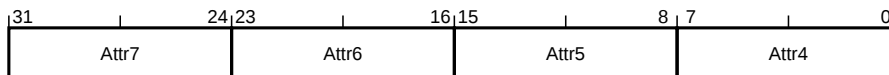
This register is banked between Security states.

#### Preface

This register is RES0 if no MPU regions are implemented in the corresponding Security state.

#### Field descriptions

The MPU\_MAIR1 bit assignments are:



#### Attr $m$ , bits [8( $m-4$ )+7:8( $m-4$ )], for $m = 4$ to 7

Attribute  $m$ . Memory attribute encoding for MPU regions with an AttrIdx of  $m$ .

The possible values of this field are:

#### All

See MAIR\_ATTR for encoding.

This field resets to an Architecturally UNKNOWN value on a Warm reset.



All other values are reserved.

For any type of Device memory, the value of this field is ignored.

This field resets to an **UNKNOWN** value on a Warm reset.

**AP[2:1], bits [2:1]**

Access permissions. Defines the access permissions for this region.

The possible values of this field are:

**0b00**

Read/write by privileged code only.

**0b01**

Read/write by any privilege level.

**0b10**

Read-only by privileged code only.

**0b11**

Read-only by any privilege level.

This field resets to an **UNKNOWN** value on a Warm reset.

**XN, bit [0]**

Execute Never. Defines whether code can be executed from this region.

The possible values of this bit are:

**0**

Execution only permitted if read permitted.

**1**

Execution not permitted.

This bit resets to an **UNKNOWN** value on a Warm reset.

### D1.2.151 MPU\_RBAR\_An, MPU Region Base Address Register Alias, n = 1 - 3

The MPU\_RBAR\_A{1..3} characteristics are:

**Purpose**

Provides indirect read and write access to the base address of the MPU region selected by MPU\_RNR[7:2]:(n[1:0]) for the selected Security state.

**Usage constraints**

Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

**Configurations**

Present only if the Main Extension is implemented.

This register is RES0 if the Main Extension is not implemented.

**Attributes**

32-bit read/write register located at 0xE000EDA4 + 8(n-1).

To allow lock down of this register it is IMPLEMENTATION DEFINED whether this register is writable.

Secure software can access the Non-secure version of this register via MPU\_RBAR\_An\_NS located at 0xE002EDA4 + 8(n-1). The location 0xE002EDA4 + 8(n-1) is RES0 to software executing in Non-secure state and the debugger.

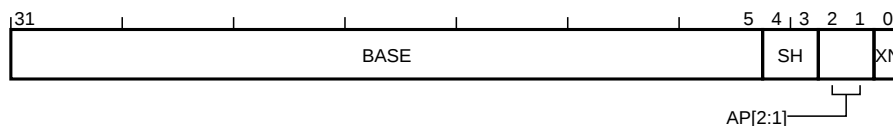
This register is banked between Security states.

**Preface**

This register is an alias of the MPU\_RBAR register and provides access to the configuration of the MPU region selected by MPU\_RNR.REGION had REGION[1:0] been set to n[1:0].

**Field descriptions**

The MPU\_RBAR\_A{1..3} bit assignments are:



**BASE, bits [31:5]**

Base address. Contains bits [31:5] of the lower inclusive limit of the selected MPU memory region. This value is zero extended to provide the base address to be checked against.

It is IMPLEMENTATION DEFINED whether any of the BASE bits are WI

This field resets to an Architecturally UNKNOWN value on a Warm reset.

**SH, bits [4:3]**

Shareability. Defines the Shareability domain of this region for Normal memory.

The possible values of this field are:

**0b00**

Non-shareable.

**0b10**

Outer Shareable.

**0b11**

Inner Shareable.

All other values are reserved.

For any type of Device memory, the value of this field is ignored.

This field resets to an **UNKNOWN** value on a Warm reset.

**AP[2:1], bits [2:1]**

Access permissions. Defines the access permissions for this region.

The possible values of this field are:

**0b00**

Read/write by privileged code only.

**0b01**

Read/write by any privilege level.

**0b10**

Read-only by privileged code only.

**0b11**

Read-only by any privilege level.

This field resets to an **UNKNOWN** value on a Warm reset.

**XN, bit [0]**

Execute Never. Defines whether code can be executed from this region.

The possible values of this bit are:

**0**

Execution only permitted if read permitted.

**1**

Execution not permitted.

This bit resets to an **UNKNOWN** value on a Warm reset.

## D1.2.152 MPU\_RLAR, MPU Region Limit Address Register

The MPU\_RLAR characteristics are:

### Purpose

Provides indirect read and write access to the limit address of the currently selected MPU region for the selected Security state.

### Usage constraints

Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are **UNPREDICTABLE**.

### Configurations

This register is always implemented.

### Attributes

32-bit read/write register located at 0xE000EDA0.

To allow lock down of this register it is **IMPLEMENTATION DEFINED** whether this register is writable.

Secure software can access the Non-secure version of this register via MPU\_RLAR\_NS located at 0xE002EDA0. The location 0xE002EDA0 is **RES0** to software executing in Non-secure state and the debugger.

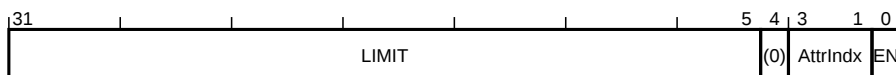
This register is banked between Security states.

### Preface

This register provides access to the configuration of the MPU region selected by MPU\_RNR.REGION for the appropriate Security state. The field descriptions apply to the currently selected region.

### Field descriptions

The MPU\_RLAR bit assignments are:



#### LIMIT, bits [31:5]

Limit address. Contains bits [31:5] of the upper inclusive limit of the selected MPU memory region. This value is postfixed with 0x1F to provide the limit address to be checked against.

It is **IMPLEMENTATION DEFINED** whether any of the LIMIT bits are WI

This field resets to an **Architecturally UNKNOWN** value on a Warm reset.

#### Bit [4]

Reserved, **RES0**.

#### AttrIdx, bits [3:1]

Attribute index. Associates a set of attributes in the MPU\_MAIRO and MPU\_MAIR1 fields.

This field resets to an **UNKNOWN** value on a Warm reset.

#### EN, bit [0]

Enable. Region enable.

The possible values of this bit are:

- 0**  
Region disabled.

**1**

Region enabled.

This bit resets to zero on a Warm reset.





**0**      Region disabled.

**1**      Region enabled.

This bit resets to zero on a Warm reset.

### D1.2.154 MPU\_RNR, MPU Region Number Register

The MPU\_RNR characteristics are:

#### Purpose

Selects the region currently accessed by MPU\_RBAR and MPU\_RLAR.

#### Usage constraints

Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are **UNPREDICTABLE**.

#### Configurations

This register is always implemented.

#### Attributes

32-bit read/write register located at 0xE000ED98.

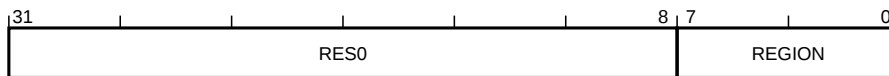
To allow lock down of this register it is **IMPLEMENTATION DEFINED** whether this register is writable.

Secure software can access the Non-secure version of this register via MPU\_RNR\_NS located at 0xE002ED98. The location 0xE002ED98 is **RES0** to software executing in Non-secure state and the debugger.

This register is banked between Security states.

### Field descriptions

The MPU\_RNR bit assignments are:



#### Bits [31:8]

Reserved, **RES0**.

#### REGION, bits [7:0]

Region number. Indicates the memory region accessed by MPU\_RBAR and MPU\_RLAR.

If no MPU regions are implemented, this field is **RES0**. Writing a value corresponding to an unimplemented region is **CONSTRAINED UNPREDICTABLE**.

This field resets to an **UNKNOWN** value on a Warm reset.

### D1.2.155 MPU\_TYPE, MPU Type Register

The MPU\_TYPE characteristics are:

#### Purpose

The MPU Type Register indicates how many regions the MPU for the selected Security state supports.

#### Usage constraints

Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

#### Configurations

This register is always implemented.

#### Attributes

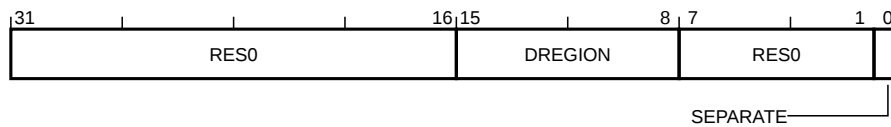
32-bit read-only register located at 0xE000ED90.

Secure software can access the Non-secure version of this register via MPU\_TYPE\_NS located at 0xE002ED90. The location 0xE002ED90 is RES0 to software executing in Non-secure state and the debugger.

This register is banked between Security states.

### Field descriptions

The MPU\_TYPE bit assignments are:



#### Bits [31:16]

Reserved, RES0.

#### DREGION, bits [15:8]

Data regions. Number of regions supported by the MPU.

If this field reads-as-zero, the PE does not implement an MPU for the selected Security state.

This field reads as an IMPLEMENTATION DEFINED value.

#### Bits [7:1]

Reserved, RES0.

#### SEPARATE, bit [0]

Separate. Indicates support for separate instructions and data address regions.

Armv8-M only supports unified MPU regions.

This bit reads as zero.

## D1.2.156 MSPLIM, Main Stack Pointer Limit Register

The MSPLIM characteristics are:

### Purpose

Holds the lower limit of the Main stack pointer.

### Usage constraints

Privileged access only. Unprivileged access is RAZ/WI, unless otherwise stated.

### Configurations

This register is always implemented.

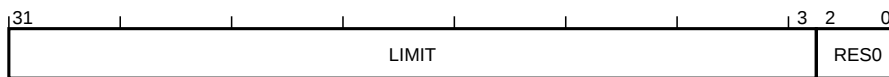
### Attributes

32-bit read/write special-purpose register.

This register is banked between Security states.

## Field descriptions

The MSPLIM bit assignments are:



### LIMIT, bits [31:3]

Stack limit. Bits [31:3] of the Main stack pointer limit address for the selected Security state.

Many instructions and exception entry will generate an exception if the appropriate stack pointer would be updated to a value lower than this limit. If the Main Extension is not implemented, the Non-secure MSPLIM is [RES0](#).

This field resets to zero on a Warm reset.

### Bits [2:0]

Reserved, [RES0](#).

## D1.2.157 MVFR0, Media and VFP Feature Register 0

The MVFR0 characteristics are:

### Purpose

Describes the features provided by the Floating-point Extension.

### Usage constraints

Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

### Configurations

Present only if the Floating-point Extension is implemented.

This register is RES0 if the Floating-point Extension is not implemented.

### Attributes

32-bit read-only register located at 0xE000EF40.

Secure software can access the Non-secure version of this register via MVFR0\_NS located at 0xE002EF40. The location 0xE002EF40 is RES0 to software executing in Non-secure state and the debugger.

This register is not banked between Security states.

### Preface

When the Floating-point Extension is not implemented this register reads as 0x00000000.

Where single-precision only floating-point is supported this register reads as 0x10110021.

Where single and double-precision floating-point are supported this register reads as 0x10110221.

### Field descriptions

The MVFR0 bit assignments are:

|         |       |        |          |       |       |      |         |   |
|---------|-------|--------|----------|-------|-------|------|---------|---|
| 31      | 28,27 | 24,23  | 20,19    | 16,15 | 12,11 | 8,7  | 4,3     | 0 |
| FPRound | RES0  | FPSqrt | FPDivide | RES0  | FPDP  | FPSP | SIMDReg |   |

#### FPRound, bits [31:28]

Floating-point rounding modes. Indicates the rounding modes supported by the Floating-point Extension.

The possible values of this field are:

**0b0001**

All rounding modes supported.

All other values are reserved.

This field reads as 0b0001.

#### Bits [27:24]

Reserved, RES0.

#### FPSqrt, bits [23:20]

Floating-point square root. Indicates the support for floating-point square root operations.

The possible values of this field are:

**0b0001**

Supported.

All other values are reserved.

This field reads as 0b0001.

**FPDivide, bits [19:16]**

Floating-point divide. Indicates the support for floating-point divide operations.

The possible values of this field are:

**0b0001**

Supported.

All other values are reserved.

This field reads as 0b0001.

**Bits [15:12]**

Reserved, [RES0](#).

**FPDP, bits [11:8]**

Floating-point double-precision. Indicates support for floating-point double-precision operations.

The possible values of this field are:

**0b0000**

Not supported.

**0b0010**

Supported.

All other values are reserved.

This field reads as an [IMPLEMENTATION DEFINED](#) value.

**FPSP, bits [7:4]**

Floating-point single-precision. Indicates support for floating-point single-precision operations.

The possible values of this field are:

**0b0010**

Supported.

All other values are reserved.

This field reads as 0b0010.

**SIMDReg, bits [3:0]**

SIMD registers. Indicates size of Floating-Point Extension register file.

The possible values of this field are:

**0b0001**

16 x 64-bit registers.

All other values are reserved.

This field reads as 0b0001.

## D1.2.158 MVFR1, Media and VFP Feature Register 1

The MVFR1 characteristics are:

### Purpose

Describes the features provided by the Floating-point Extension.

### Usage constraints

Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are [UNPREDICTABLE](#).

### Configurations

Present only if the Floating-point Extension is implemented.

This register is [RES0](#) if the Floating-point Extension is not implemented.

### Attributes

32-bit read-only register located at 0xE000EF44.

Secure software can access the Non-secure version of this register via MVFR1\_NS located at 0xE002EF44. The location 0xE002EF44 is [RES0](#) to software executing in Non-secure state and the debugger.

This register is not banked between Security states.

### Preface

When floating-point is not implemented this register reads as 0x00000000.

Where single-precision only floating-point is supported this register reads as 0x11000011.

Where single and double-precision floating-point are supported this register reads as 0x12000011.

### Field descriptions

The MVFR1 bit assignments are:



#### FMAC, bits [31:28]

Fused multiply accumulate. Indicates whether the Floating-point Extension implements the fused multiply accumulate instructions.

The possible values of this field are:

**0b0001**

Implemented.

All other values are reserved.

This field reads as 0b0001.

#### FPHP, bits [27:24]

Floating-point half-precision. Indicates whether the Floating-point Extension implements half-precision floating-point conversion instructions.

The possible values of this field are:

**0b0001**

Half-precision to single-precision implemented.



**0b0010**

Half-precision to single and double-precision implemented.

All other values are reserved.

This field reads as an **IMPLEMENTATION DEFINED** value.

**Bits [23:8]**

Reserved, **RES0**.

**FPDNaN, bits [7:4]**

Floating-point default NaN. Indicates whether the Floating-point Extension implementation supports NaN propagation.

The possible values of this field are:

**0b0001**

Propagation of NaN values supported.

All other values are reserved.

This field reads as 0b0001.

**FPFtZ, bits [3:0]**

Floating-point flush-to-zero. Indicates whether subnormals are always flushed-to-zero.

The possible values of this field are:

**0b0001**

Full denormalized numbers arithmetic supported.

All other values are reserved.

This field reads as 0b0001.



## D1.2.160 NSACR, Non-secure Access Control Register

The NSACR characteristics are:

### Purpose

Defines the Non-secure access permissions for both the FP Extension and coprocessors CP0 to CP7.

### Usage constraints

Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are **UNPREDICTABLE**.

### Configurations

Present only if the Main Extension is implemented.

This register is **RES0** if the Main Extension is not implemented.

### Attributes

32-bit read/write register located at 0xE000ED8C.

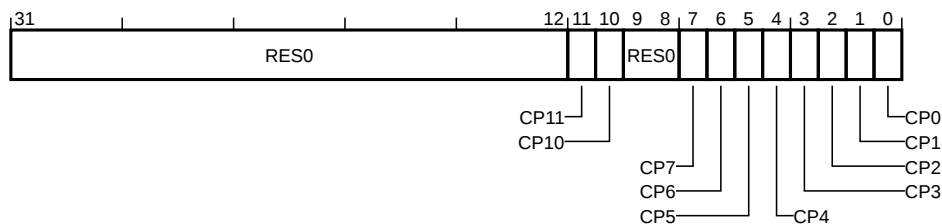
If the Security Extension is not implemented this register returns a value of 0x00000CFF.

This register is RAZ/WI when accessed as Non-secure.

This register is not banked between Security states.

## Field descriptions

The NSACR bit assignments are:



### Bits [31:12]

Reserved, **RES0**.

### CP11, bit [11]

CP11 access. Enables Non-secure access to the Floating-point Extension.

Programming with a different value than that used for CP10 is **UNPREDICTABLE**. If the Floating-point Extension is not implemented, this bit is RAZ/WI.

This bit resets to an **UNKNOWN** value on a Warm reset.

### CP10, bit [10]

CP10 access. Enables Non-secure access to the Floating-point Extension.

The possible values of this bit are:

**0**

Non-secure accesses to the Floating-point Extension generate a NOCP UsageFault.

**1**

Non-secure access to the Floating-point Extension permitted.

If the Floating-point Extension is not implemented, this bit is RAZ/WI.

This bit resets to an **UNKNOWN** value on a Warm reset.

**Bits [9:8]**

Reserved, [RES0](#).

**CPm, bit [m], for m = 0 to 7**

CPm access. Enables Non-secure access to coprocessor CPm.

The possible values of this field are:

**0**

Non-secure accesses to this coprocessor generate a NOCP UsageFault.

**1**

Non-secure access to this coprocessor permitted.

A CPm bit is RAZ/WI if CPm is:

- Not implemented.
- Not enabled for the Security state in which the PE is executing.

This field resets to an [Architecturally UNKNOWN](#) value on a Warm reset.

### D1.2.161 NVIC\_IABRn, Interrupt Active Bit Register, n = 0 - 15

The NVIC\_IABR{0..15} characteristics are:

#### Purpose

For each group of 32 interrupts, shows the active state of each interrupt.

#### Usage constraints

Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

#### Configurations

This register is always implemented.

#### Attributes

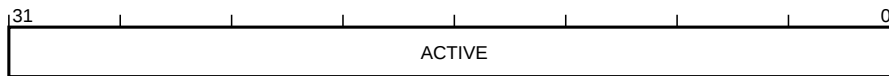
32-bit read-only register located at  $0xE000E300 + 4n$ .

Secure software can access the Non-secure version of this register via NVIC\_IABRn\_NS located at  $0xE002E300 + 4n$ . The location  $0xE002E300 + 4n$  is RESO to software executing in Non-secure state and the debugger.

This register is not banked between Security states.

### Field descriptions

The NVIC\_IABR{0..15} bit assignments are:



#### ACTIVE, bits [31:0]

Active state. For ACTIVE[m] in NVIC\_IABRn, indicates the active state for interrupt  $32n+m$ .

The possible values of each bit are:

**0**

Interrupt not active.

**1**

Interrupt is active.

Bits corresponding to unimplemented interrupts are RES0. Bits corresponding to interrupts targeting Secure state are RAZ/WI from Non-secure.

This field resets to zero on a Warm reset.

## D1.2.162 NVIC\_ICERn, Interrupt Clear Enable Register, n = 0 - 15

The NVIC\_ICER{0..15} characteristics are:

### Purpose

Clears or reads the enabled state of each group of 32 interrupts.

### Usage constraints

Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

### Configurations

This register is always implemented.

### Attributes

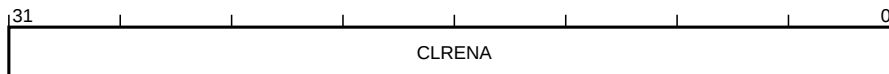
32-bit read/write-one-to-clear register located at  $0 \times E000E180 + 4n$ .

Secure software can access the Non-secure version of this register via NVIC\_ICERn\_NS located at  $0 \times E002E180 + 4n$ . The location  $0 \times E002E180 + 4n$  is RESO to software executing in Non-secure state and the debugger.

This register is not banked between Security states.

## Field descriptions

The NVIC\_ICER{0..15} bit assignments are:



### CLRENA, bits [31:0], on a write

Clear enable. For CLRENA[m] in NVIC\_ICERn, allows interrupt  $32n + m$  to be disabled.

The possible values of each bit are:

**0**

No effect.

**1**

Disable interrupt  $32n + m$ .

Bits corresponding to unimplemented interrupts are RESO. Bits corresponding to interrupts targeting Secure state are RAZ/WI from Non-secure.

### CLRENA, bits [31:0], on a read

Clear enable. For CLRENA[m] in NVIC\_ICERn, indicates whether interrupt  $32n + m$  is enabled.

The possible values of each bit are:

**0**

Interrupt  $32n + m$  disabled.

**1**

Interrupt  $32n + m$  enabled.

Bits corresponding to unimplemented interrupts are RESO. Bits corresponding to interrupts targeting Secure state are RAZ/WI from Non-secure.

This field resets to zero on a Warm reset.

### D1.2.163 NVIC\_ICPRn, Interrupt Clear Pending Register, n = 0 - 15

The NVIC\_ICPR{0..15} characteristics are:

#### Purpose

Clears or reads the pending state of each group of 32 interrupts.

#### Usage constraints

Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

#### Configurations

This register is always implemented.

#### Attributes

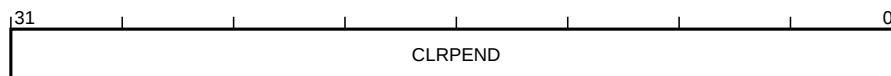
32-bit read/write-one-to-clear register located at  $0 \times E000E280 + 4n$ .

Secure software can access the Non-secure version of this register via NVIC\_ICPRn\_NS located at  $0 \times E002E280 + 4n$ . The location  $0 \times E002E280 + 4n$  is RESO to software executing in Non-secure state and the debugger.

This register is not banked between Security states.

### Field descriptions

The NVIC\_ICPR{0..15} bit assignments are:



#### CLRPEND, bits [31:0], on a write

Clear pending. For CLRPEND[m] in NVIC\_ICPRn, allows interrupt  $32n + m$  to be unpended.

The possible values of each bit are:

**0**

No effect.

**1**

Clear pending state of interrupt  $32n + m$ .

Bits corresponding to unimplemented interrupts are RESO. Bits corresponding to interrupts targeting Secure state are RAZ/WI from Non-secure.

#### CLRPEND, bits [31:0], on a read

Clear pending. For CLRPEND[m] in NVIC\_ICPRn, indicates whether interrupt  $32n + m$  is pending.

The possible values of each bit are:

**0**

Interrupt  $32n + m$  is not pending.

**1**

Interrupt  $32n + m$  is pending.

Bits corresponding to unimplemented interrupts are RESO. Bits corresponding to interrupts targeting Secure state are RAZ/WI from Non-secure.

## D1.2.164 NVIC\_IPRn, Interrupt Priority Register, n = 0 - 123

The NVIC\_IPR{0..123} characteristics are:

### Purpose

Sets or reads interrupt priorities.

### Usage constraints

Privileged access permitted only. Unprivileged accesses generate a fault.

If the Main Extension is implemented, this register is word, halfword, and byte accessible.

If the Main Extension is not implemented, this register is word accessible only, halfword and byte accesses are **UNPREDICTABLE**.

### Configurations

This register is always implemented.

### Attributes

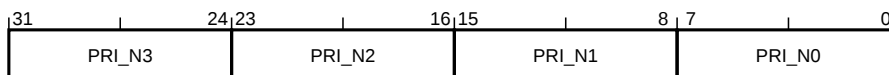
32-bit read/write register located at  $0xE000E400 + 4n$ .

Secure software can access the Non-secure version of this register via NVIC\_IPRn\_NS located at  $0xE002E400 + 4n$ . The location  $0xE002E400 + 4n$  is **RES0** to software executing in Non-secure state and the debugger.

This register is not banked between Security states.

## Field descriptions

The NVIC\_IPR{0..123} bit assignments are:



### PRI\_Nm, bits [8m+7:8m], for m = 0 to 3

Priority 'N'+m. For register NVIC\_IPRn, this field indicates and allows modification of the priority of interrupt number  $4n+m$ , or is **RES0** if the PE does not implement this interrupt.

If the PE implements fewer than 8 bits of priority, then the least significant bits of this field are **RES0**.

If interrupt number  $4n+m$  targets Secure state, this field is **RAZ/WI** from Non-secure.

This field resets to zero on a Warm reset.



## D1.2.165 NVIC\_ISERn, Interrupt Set Enable Register, n = 0 - 15

The NVIC\_ISER{0..15} characteristics are:

### Purpose

Enables or reads the enabled state of each group of 32 interrupts.

### Usage constraints

Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

### Configurations

This register is always implemented.

### Attributes

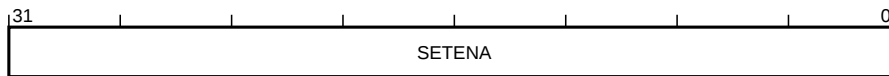
32-bit read/write-one-to-set register located at  $0xE000E100 + 4n$ .

Secure software can access the Non-secure version of this register via NVIC\_ISERn\_NS located at  $0xE002E100 + 4n$ . The location  $0xE002E100 + 4n$  is RESO to software executing in Non-secure state and the debugger.

This register is not banked between Security states.

## Field descriptions

The NVIC\_ISER{0..15} bit assignments are:



### SETENA, bits [31:0], on a write

Set enable. For SETENA[m] in NVIC\_ISERn, allows interrupt  $32n + m$  to be set enabled.

The possible values of each bit are:

**0**

No effect.

**1**

Enable interrupt  $32n + m$ .

Bits corresponding to unimplemented interrupts are RESO. Bits corresponding to interrupts targeting Secure state are RAZ/WI from Non-secure.

This field resets to zero on a Warm reset.

### SETENA, bits [31:0], on a read

Set enable. For SETENA[m] in NVIC\_ISERn, indicates whether interrupt  $32n + m$  is enabled.

The possible values of each bit are:

**0**

Interrupt  $32n + m$  disabled.

**1**

Interrupt  $32n + m$  enabled.

Bits corresponding to unimplemented interrupts are RESO. Bits corresponding to interrupts targeting Secure state are RAZ/WI from Non-secure.

This field resets to zero on a Warm reset.

## D1.2.166 NVIC\_ISPRn, Interrupt Set Pending Register, n = 0 - 15

The NVIC\_ISPR{0..15} characteristics are:

### Purpose

Enables or reads the pending state of each group of 32 interrupts.

### Usage constraints

Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

### Configurations

This register is always implemented.

### Attributes

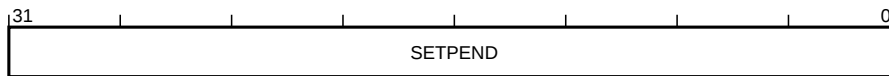
32-bit read/write-one-to-set register located at  $0xE000E200 + 4n$ .

Secure software can access the Non-secure version of this register via NVIC\_ISPRn\_NS located at  $0xE002E200 + 4n$ . The location  $0xE002E200 + 4n$  is RESO to software executing in Non-secure state and the debugger.

This register is not banked between Security states.

## Field descriptions

The NVIC\_ISPR{0..15} bit assignments are:



### SETPEND, bits [31:0], on a write

Set pending. For SETPEND[m] in NVIC\_ISPRn, allows interrupt  $32n + m$  to be set pending.

The possible values of each bit are:

**0**

No effect.

**1**

Pend interrupt  $32n + m$ .

Bits corresponding to unimplemented interrupts are RESO. Bits corresponding to interrupts targeting Secure state are RAZ/WI from Non-secure.

This field is write-one-to-set. Writes of zero are ignored.

This field resets to zero on a Warm reset.

### SETPEND, bits [31:0], on a read

Set pending. For SETPEND[m] in NVIC\_ISPRn, indicates whether interrupt  $32n + m$  is pending.

The possible values of each bit are:

**0**

Interrupt  $32n + m$  is not pending.

**1**

Interrupt  $32n + m$  pending.

Bits corresponding to unimplemented interrupts are [RES0](#). Bits corresponding to interrupts targeting Secure state are RAZ/WI from Non-secure.

This field resets to zero on a Warm reset.

### D1.2.167 NVIC\_ITNSn, Interrupt Target Non-secure Register, n = 0 - 15

The NVIC\_ITNS{0..15} characteristics are:

#### Purpose

For each group of 32 interrupts, determines whether each interrupt targets Non-secure or Secure state.

#### Usage constraints

Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

#### Configurations

This register is always implemented.

#### Attributes

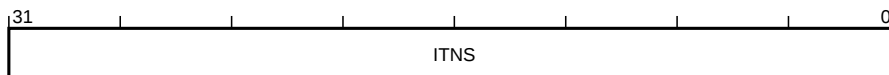
32-bit read/write register located at  $0 \times E000E380 + 4n$ .

This register is RAZ/WI when accessed as Non-secure.

This register is not banked between Security states.

### Field descriptions

The NVIC\_ITNS{0..15} bit assignments are:



#### ITNS, bits [31:0]

Interrupt Targets Non-secure. For ITNS[m] in NVIC\_ITNSn, this field indicates and allows modification of the target Security state for interrupt  $32n+m$ .

The possible values of each bit are:

**0**

Interrupt targets Secure state.

**1**

Interrupt targets Non-secure state.

Bits corresponding to unimplemented interrupts are RES0. It is IMPLEMENTATION DEFINED whether individual bits are WI and have an IMPLEMENTATION DEFINED constant value. Where an interrupt is configured to target Secure state, accesses to the associated fields in Non-secure versions of the NVIC\_IABR, NVIC\_ICER, NVIC\_ISER, NVIC\_ICPR, NVIC\_IPR and NVIC\_ISPR are RAZ/WI.

This field resets to zero on a Warm reset.

### D1.2.168 PC, Program Counter

The PC characteristics are:

**Purpose**

Holds the current Program Counter value.

**Usage constraints**

Privileged and unprivileged access permitted.

**Configurations**

This register is always implemented.

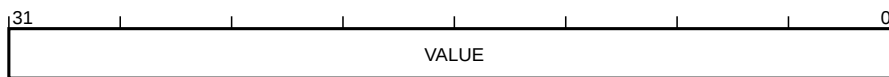
**Attributes**

32-bit read/write special-purpose register.

This register is not banked between Security states.

### Field descriptions

The PC bit assignments are:



**VALUE, bits [31:0]**

Program Counter. Holds the address of the current instruction.

Software can refer to PC as R15.

This field resets to an **UNKNOWN** value on a Warm reset.



### D1.2.170 PSPLIM, Process Stack Pointer Limit Register

The PSPLIM characteristics are:

**Purpose**

Holds the lower limit for the Process stack pointer.

**Usage constraints**

Privileged access only. Unprivileged access is RAZ/WI, unless otherwise stated.

**Configurations**

This register is always implemented.

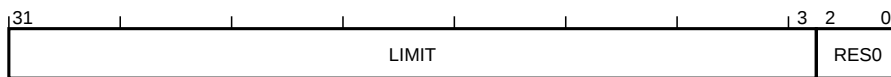
**Attributes**

32-bit read/write special-purpose register.

This register is banked between Security states.

### Field descriptions

The PSPLIM bit assignments are:



**LIMIT, bits [31:3]**

Stack limit. Bits [31:3] of the Process stack limit address for the selected Security state.

Many instructions and exception entry will generate an exception if the appropriate stack pointer would be updated to a value lower than this limit. If the Main Extension is not implemented, the Non-secure PSPLIM is [RES0](#).

This field resets to zero on a Warm reset.

**Bits [2:0]**

Reserved, [RES0](#).

### D1.2.171 Rn, General-Purpose Register, n = 0 - 12

The R{0..12} characteristics are:

**Purpose**

General-purpose register.

**Usage constraints**

Both privileged and unprivileged accesses are permitted.

This register is word, halfword, and byte accessible.

**Configurations**

This register is always implemented.

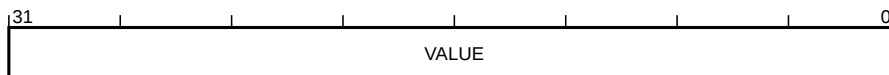
**Attributes**

32-bit read/write register.

This register is not banked between Security states.

### Field descriptions

The R{0..12} bit assignments are:



**VALUE, bits [31:0]**

General purpose register value. Armv8-M implemented thirteen general-purpose 32-bit registers, R0 to R12.

This field resets to an **UNKNOWN** value on a Warm reset.



## D1.2.172 RETPSR, Combined Exception Return Program Status Registers

The RETPSR characteristics are:

### Purpose

Value pushed to the stack on exception entry. On exception return this is used to restore the flags and other architectural state. This payload is also used for FNC\_RETURN stacking, however in this case only some of the fields are used. See FunctionReturn() for details.

### Usage constraints

None.

### Configurations

All.

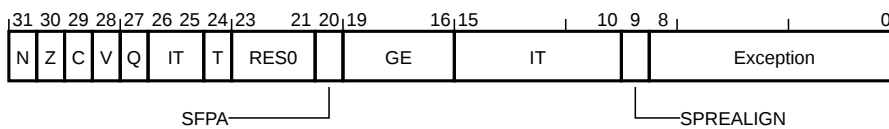
### Attributes

32-bit payload.

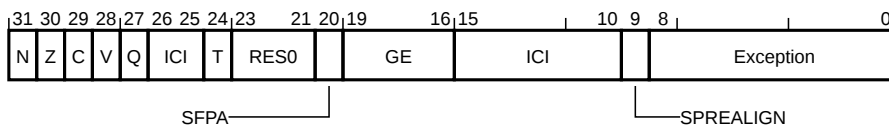
### Field descriptions

The RETPSR bit assignments are:

When {RETPSR[26:25], RETPSR[11:10]} != 0:



When {RETPSR[26:25], RETPSR[11:10]} == 0:



### N, bit [31]

Negative flag. Value corresponding to APSR.N.

### Z, bit [30]

Zero flag. Value corresponding to APSR.Z.

### C, bit [29]

Carry flag. Value corresponding to APSR.C.

### V, bit [28]

Overflow flag. Value corresponding to APSR.V.

### Q, bit [27]

Saturate flag. Value corresponding to APSR.Q.

### T, bit [24]

T32 state. Value corresponding to EPSR.T.

### Bits [23:21]

Reserved, RES0.

### SFPA, bit [20]

Secure floating-point active. Value corresponding to CONTROL.SFPA.

**GE, bits [19:16]**

Greater-than or equal flag. Value corresponding to APSR.GE.

**IT, bits [15:10,26:25]** , when [{RETPSR[26:25], RETPSR[11:10]} != 0]

If-then flags. Value corresponding to EPSR.IT.

**ICI, bits [26:25,15:10]** , when [{RETPSR[26:25], RETPSR[11:10]} == 0]

Interrupt continuation flags. Value corresponding to EPSR.ICI.

**SPREALIGN, bit [9]**

Stack-pointer re-align. Indicates whether the SP was re-aligned to an 8-byte alignment on exception entry.

The possible values of this bit are:

**0**

The stack pointer was 8-byte aligned before exception entry began, no special handling is required on exception return.

**1**

The stack pointer was only 4-byte aligned before exception entry. The exception entry realigned SP to 8-byte alignment by increasing the stack frame size by 4-bytes.

**Exception, bits [8:0]**

Exception number. Value corresponding to IPSR.Exception.

### D1.2.173 SAU\_CTRL, SAU Control Register

The SAU\_CTRL characteristics are:

#### Purpose

Allows enabling of the Security Attribution Unit.

#### Usage constraints

Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

#### Configurations

This register is always implemented.

#### Attributes

32-bit read/write register located at 0xE000EDD0.

To allow lock down of this register it is IMPLEMENTATION DEFINED whether this register is writable.

This register is RAZ/WI when accessed as Non-secure.

This register is not banked between Security states.

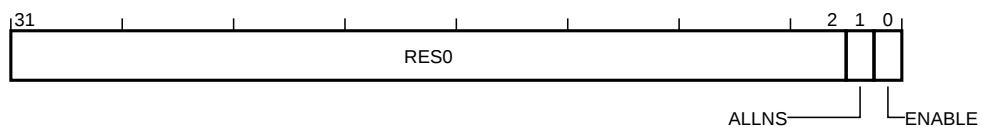
#### Preface

It is IMPLEMENTATION DEFINED whether this register:

- Resets to 0x0 - in this case SAU\_REGIONn registers are UNKNOWN at reset.
- Resets to an IMPLEMENTATION DEFINED value.

#### Field descriptions

The SAU\_CTRL bit assignments are:



#### Bits [31:2]

Reserved, RES0.

#### ALLNS, bit [1]

All Non-secure. When SAU\_CTRL.ENABLE is 0 this bit controls if the memory is marked as Non-secure or Secure.

The possible values of this bit are:

**0**

Memory is marked as Secure and is not Non-secure callable.

**1**

Memory is marked as Non-secure.

This bit resets to an IMPLEMENTATION DEFINED value on a Warm reset.

#### ENABLE, bit [0]

Enable. Enables the SAU.

The possible values of this bit are:

**0** The SAU is disabled.

**1** The SAU is enabled.

If this register resets to 1, the SAU region registers also reset to an **IMPLEMENTATION DEFINED** value.

This bit resets to an **IMPLEMENTATION DEFINED** value on a Warm reset.

### D1.2.174 SAU\_RBAR, SAU Region Base Address Register

The SAU\_RBAR characteristics are:

#### Purpose

Provides indirect read and write access to the base address of the currently selected SAU region.

#### Usage constraints

Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are [UNPREDICTABLE](#).

#### Configurations

This register is always implemented.

#### Attributes

32-bit read/write register located at 0xE000EDDC.

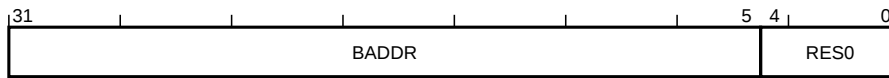
To allow lock down of this register it is [IMPLEMENTATION DEFINED](#) whether this register is writable.

This register is RAZ/WI when accessed as Non-secure.

This register is not banked between Security states.

### Field descriptions

The SAU\_RBAR bit assignments are:



#### BADDR, bits [31:5]

Base address. Holds bits [31:5] of the base address for the selected SAU region.

Bits [4:0] of the base address are defined as 0x00.

This field resets to an [IMPLEMENTATION DEFINED](#) value on a Warm reset.

#### Bits [4:0]

Reserved, [RES0](#).

### D1.2.175 SAU\_RLAR, SAU Region Limit Address Register

The SAU\_RLAR characteristics are:

#### Purpose

Provides indirect read and write access to the limit address of the currently selected SAU region.

#### Usage constraints

Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are **UNPREDICTABLE**.

#### Configurations

This register is always implemented.

#### Attributes

32-bit read/write register located at 0xE000EDE0.

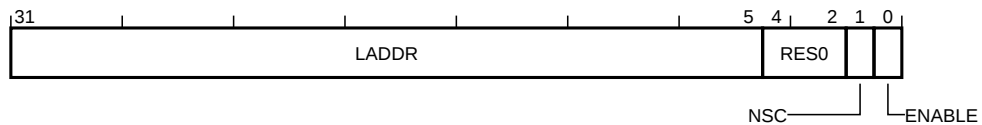
To allow lock down of this register it is **IMPLEMENTATION DEFINED** whether this register is writable.

This register is RAZ/WI when accessed as Non-secure.

This register is not banked between Security states.

### Field descriptions

The SAU\_RLAR bit assignments are:



#### LADDR, bits [31:5]

Limit address. Holds bits [31:5] of the limit address for the selected SAU region.

Bits [4:0] of the limit address are defined as 0x1F.

This field resets to an **IMPLEMENTATION DEFINED** value on a Warm reset.

#### Bits [4:2]

Reserved, **RES0**.

#### NSC, bit [1]

Non-secure callable. Controls whether Non-secure state is permitted to execute an SG instruction from this region.

The possible values of this bit are:

**0**

Region is not Non-secure callable.

**1**

Region is Non-secure callable.

This bit resets to an **IMPLEMENTATION DEFINED** value on a Warm reset.

#### ENABLE, bit [0]

Enable. SAU region enable.

The possible values of this bit are:

**0**

SAU region is disabled.

**1**

SAU region is enabled.

This bit resets to an **IMPLEMENTATION DEFINED** value on a Warm reset.





### D1.2.177 SAU\_TYPE, SAU Type Register

The SAU\_TYPE characteristics are:

#### Purpose

Indicates the number of regions implemented by the Security Attribution Unit.

#### Usage constraints

Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

#### Configurations

This register is always implemented.

#### Attributes

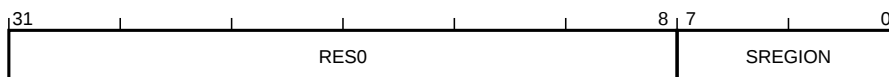
32-bit read-only register located at 0xE000EDD4.

This register is RAZ/WI when accessed as Non-secure.

This register is not banked between Security states.

### Field descriptions

The SAU\_TYPE bit assignments are:



#### Bits [31:8]

Reserved, RES0.

#### SREGION, bits [7:0]

SAU regions. The number of implemented SAU regions.

If this field is RAZ, the SAU behaves as follows:

- SAU\_CTRL.ENABLE behaves as RAZ/WI.
- It is IMPLEMENTATION DEFINED whether SAU\_CTRL.ALLNS behaves as RAO/WI and all attribution is performed by the IDAU.
- SAU\_RNR, SAU\_RBAR, and SAU\_RLAR behave as RAZ/WI.

This field reads as an IMPLEMENTATION DEFINED value.

## D1.2.178 SCR, System Control Register

The SCR characteristics are:

### Purpose

Sets or returns system control data.

### Usage constraints

Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are **UNPREDICTABLE**.

### Configurations

This register is always implemented.

### Attributes

32-bit read/write register located at 0xE000ED10.

Secure software can access the Non-secure version of this register via SCR\_NS located at 0xE002ED10. The location 0xE002ED10 is **RES0** to software executing in Non-secure state and the debugger.

This register is banked between Security states on a bit by bit basis.

## Field descriptions

The SCR bit assignments are:



### Bits [31:5]

Reserved, **RES0**.

### SEVONPEND, bit [4]

Send event on pend. Determines whether an interrupt assigned to the same Security state as the SEVONPEND bit transitioning from inactive state to pending state generates a wakeup event.

This bit is banked between Security states.

The possible values of this bit are:

**0**

Transitions from inactive to pending are not wakeup events.

**1**

Transitions from inactive to pending are wakeup events.

This bit resets to zero on a Warm reset.

### SLEEPDEEPS, bit [3]

Sleep deep secure. This field controls whether the SLEEPDEEP bit is only accessible from the Secure state.

This bit is not banked between Security states.

The possible values of this bit are:

**0**

The SLEEPDEEP bit accessible from both Security states.

**1**

The SLEEPDEEP bit behaves as RAZ/WI when accessed from the Non-secure state.

This bit is only accessible from the Secure state, and behaves as RAZ/WI when accessed from the Non-secure state. If a PE does not implement the deep sleep state this bit behaves as RAZ/WI from both Security states.

This bit is RAZ/WI from Non-secure state.

This bit resets to zero on a Warm reset.

**SLEEPDEEP, bit [2]**

Sleep deep. Provides a qualifying hint indicating that waking from sleep might take longer. An implementation can use this bit to select between two alternative sleep states.

This bit is not banked between Security states.

The possible values of this bit are:

**0**

Selected sleep state is not deep sleep.

**1**

Selected sleep state is deep sleep.

Details of the implemented sleep states, if any, and details of the use of this bit, are [IMPLEMENTATION DEFINED](#). If the PE does not implement a deep sleep state then this bit can be RAZ/WI.

This bit resets to zero on a Warm reset.

**SLEEPONEXIT, bit [1]**

Sleep on exit. Determines whether, on an exit from an ISR that returns to the base level of execution priority, the PE enters a sleep state.

This bit is banked between Security states.

The possible values of this bit are:

**0**

Enter sleep state disabled.

**1**

Enter sleep state permitted.

The Secure version of this field is used if the Background state being returned to is the Secure state, otherwise the Non-secure version is used.

This bit resets to zero on a Warm reset.

**Bit [0]**

Reserved, [RES0](#).

### D1.2.179 SFAR, Secure Fault Address Register

The SFAR characteristics are:

#### Purpose

Shows the address of the memory location that caused a Security violation.

#### Usage constraints

Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are [UNPREDICTABLE](#).

#### Configurations

Present only if the Main Extension is implemented.

This register is [RES0](#) if the Main Extension is not implemented.

#### Attributes

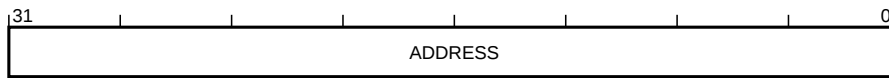
32-bit read/write register located at 0xE000EDE8.

This register is RAZ/WI when accessed as Non-secure.

This register is not banked between Security states.

### Field descriptions

The SFAR bit assignments are:



#### ADDRESS, bits [31:0]

Address. The address of an access that caused an attribution unit violation. This field is only valid when SFSR.SFARVALID is set. This allows the actual flip flops associated with this register to be shared with other fault address registers. If an implementation chooses to share the storage in this way, care must be taken to not leak Secure address information to the Non-secure state. One way of achieving this is to share the SFAR register with the MMFAR\_S register, which is not accessible to the Non-secure state.

This field resets to an [UNKNOWN](#) value on a Warm reset.

## D1.2.180 SFSR, Secure Fault Status Register

The SFSR characteristics are:

### Purpose

Provides information about any security related faults.

### Usage constraints

Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are **UNPREDICTABLE**.

### Configurations

Present only if the Main Extension is implemented.

This register is **RES0** if the Main Extension is not implemented.

### Attributes

32-bit read/write-one-to-clear register located at 0xE000EDE4.

This register is RAZ/WI when accessed as Non-secure.

This register is not banked between Security states.

## Field descriptions

The SFSR bit assignments are:



### Bits [31:8]

Reserved, **RES0**.

### LSERR, bit [7]

Lazy state error flag. Sticky flag indicating that an error occurred during lazy state activation or deactivation.

The possible values of this bit are:

**0**  
Error has not occurred.

**1**  
Error has occurred.

This bit resets to zero on a Warm reset.

### SFARVALID, bit [6]

Secure fault address valid. This bit is set when the SFAR register contains a valid value. As with similar fields, such as BFSR.BFARVALID and MMFSR.MMARVALID, this bit can be cleared by other exceptions, such as BusFault.

The possible values of this bit are:

**0**  
SFAR content not valid.

**1**  
SFAR content valid.

This bit resets to zero on a Warm reset.

**LSPERR, bit [5]**

Lazy state preservation error flag. Sticky flag indicating that an SAU or IDAU violation occurred during the lazy preservation of floating-point state.

The possible values of this bit are:

**0**  
Error has not occurred.

**1**  
Error has occurred.

This bit resets to zero on a Warm reset.

**INVTRAN, bit [4]**

Invalid transition flag. Sticky flag indicating that an exception was raised due to a branch that was not flagged as being domain crossing causing a transition from Secure to Non-secure memory.

The possible values of this bit are:

**0**  
Error has not occurred.

**1**  
Error has occurred.

This bit resets to zero on a Warm reset.

**AUVIOL, bit [3]**

Attribution unit violation flag.

Sticky flag indicating that an attempt was made to access parts of the address space that are marked as Secure with NS-Req for the transaction set to Non-secure.

This bit is not set if the violation occurred during:

- Lazy state preservation, see LSPERR.
- Vector fetches.

The possible values of this bit are:

**0**  
Error has not occurred.

**1**  
Error has occurred.

This bit resets to zero on a Warm reset.

**INVER, bit [2]**

Invalid exception return flag. This can be caused by EXC\_RETURN.DCRS being set to 0 when returning from an exception in the Non-secure state, or by EXC\_RETURN.ES being set to 1 when returning from an exception in the Non-secure state.

The possible values of this bit are:

**0**  
Error has not occurred.

**1**  
Error has occurred.

This bit resets to zero on a Warm reset.

**INVIS, bit [1]**

Invalid integrity signature flag. This bit is set if the integrity signature in an exception stack frame is found to be invalid during the unstacking operation.

The possible values of this bit are:

**0**

Error has not occurred.

**1**

Error has occurred.

This bit resets to zero on a Warm reset.

**INVEP, bit [0]**

Invalid entry point. This bit is set if a function call from the Non-secure state or exception targets a non-SG instruction in the Secure state. This bit is also set if the target address is an SG instruction, but there is no matching SAU/IDAU region with the NSC flag set.

The possible values of this bit are:

**0**

Error has not occurred.

**1**

Error has occurred.

This bit resets to zero on a Warm reset.

## D1.2.181 SHCSR, System Handler Control and State Register

The SHCSR characteristics are:

### Purpose

Provides access to the active and pending status of system exceptions.

### Usage constraints

Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are **UNPREDICTABLE**.

### Configurations

This register is always implemented.

### Attributes

32-bit read/write register located at 0xE000ED24.

Secure software can access the Non-secure version of this register via SHCSR\_NS located at 0xE002ED24. The location 0xE002ED24 is **RES0** to software executing in Non-secure state and the debugger.

This register is banked between Security states on a bit by bit basis.

## Preface

Exception processing automatically updates the SHCSR fields. However, software can write to the register to add or remove the pending or active state of an exception. When updating the SHCSR, Arm recommends using a read-modify-write sequence, to avoid unintended effects on the state of the exception handlers.

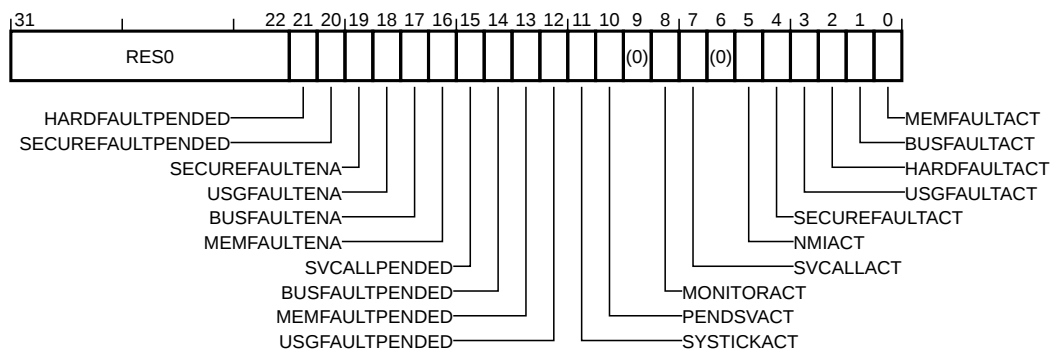
Removing the active state of an exception can change the current execution priority, and affect the exception return consistency checks. If software removes the active state, causing a change in current execution priority, this can defeat the architectural behavior that prevents an exception from preempting its own handler.

Pending state bits are set to one when an exception occurs and are cleared to zero when the exception becomes active.

Active state bits are set to one when the associated exception becomes active.

## Field descriptions

The SHCSR bit assignments are:



### Bits [31:22]

Reserved, **RES0**.



**HARDFaultPENDED, bit [21]**

HardFault exception pended state. This bit indicates and allows modification of the pending state of the HardFault exception corresponding to the selected Security state.

This bit is banked between Security states.

The possible values of this bit are:

**0**

HardFault exception not pending for the selected Security state.

**1**

HardFault exception pending for the selected Security state.

The Non-secure view of this bit is RAZ/WI if AIRCR.BFHFNMINs is zero.

This bit resets to zero on a Warm reset.

**Note**

The Non-secure HardFault exception will not preempt if AIRCR.BFHFNMINs is set to zero.

**SECUREFaultPENDED, bit [20]**

SecureFault exception pended state. This bit indicates and allows modification of the pending state of the SecureFault exception.

This bit is not banked between Security states.

The possible values of this bit are:

**0**

SecureFault exception not pending.

**1**

SecureFault exception pending.

This bit is RAZ/WI from Non-secure state.

If the Main Extension is not implemented, this bit is [RES0](#).

This bit resets to zero on a Warm reset.

**SECUREFaultENA, bit [19]**

SecureFault exception enable. The value of this bit defines whether the SecureFault exception is enabled.

This bit is not banked between Security states.

The possible values of this bit are:

**0**

SecureFault exception disabled.

**1**

SecureFault exception enabled.

When disabled, exceptions that target SecureFault escalate to Secure state HardFault.

This bit is RAZ/WI from Non-secure state.

If the Main Extension is not implemented, this bit is [RES0](#).

This bit resets to zero on a Warm reset.

**USGFaultENA, bit [18]**

UsageFault exception enable. The value of this bit defines whether the UsageFault exception is enabled for the selected Security state.

This bit is banked between Security states.

The possible values of this bit are:

**0**

UsageFault exception disabled for the selected Security state.

**1**

UsageFault exception enabled for the selected Security state.

When the UsageFault exception is disabled, exceptions targeting UsageFault escalate to HardFault.

If the Main Extension is not implemented, this bit is **RES0**.

This bit resets to zero on a Warm reset.

#### **BUSFAULTENA, bit [17]**

BusFault exception enable. The value of this bit defines whether the BusFault exception is enabled.

This bit is not banked between Security states.

The possible values of this bit are:

**0**

BusFault exception disabled.

**1**

BusFault exception enabled.

The BusFault exception is not banked between Security states. When the BusFault exception is disabled, exceptions targeting BusFault escalate to HardFault.

If AIRCR.BFHFNMINS is zero this bit is RAZ/WI from Non-secure state.

If the Main Extension is not implemented, this bit is **RES0**.

This bit resets to zero on a Warm reset.

#### **MEMFAULTENA, bit [16]**

MemManage exception enable. The value of this bit defines whether the MemManage exception is enabled for the selected Security state.

This bit is banked between Security states.

The possible values of this bit are:

**0**

MemManage exception disabled for the selected Security state.

**1**

MemManage exception enabled for the selected Security state.

If the Main Extension is not implemented, this bit is **RES0**.

This bit resets to zero on a Warm reset.

#### **Note**

When the MemManage exception is disabled, exceptions targeting MemManage escalate to Hard-Fault.

#### **SVCALLPENDEd, bit [15]**

SVCAll exception pending state. This bit indicates and allows modification of the pending state of the SVCAll exception for the selected Security state.

This bit is banked between Security states.

The possible values of this bit are:

**0**  
SVCAll exception not pending for the selected Security state.

**1**  
SVCAll exception pending for the selected Security state.

This bit resets to zero on a Warm reset.

**BUSFAULTPENDED, bit [14]**

BusFault exception pended state. This bit indicates and allows modification of the pending state of the BusFault exception.

This bit is not banked between Security states.

The possible values of this bit are:

**0**  
BusFault exception not pending.

**1**  
BusFault exception pending.

The BusFault exception is not banked between Security states.

If AIRCR.BFHFNMIN is zero this bit is RAZ/WI from Non-secure state.

If the Main Extension is not implemented, this bit is **RES0**.

This bit resets to zero on a Warm reset.

**MEMFAULTPENDED, bit [13]**

MemManage exception pended state. This bit indicates and allows modification of the pending state of the MemManage exception for the selected Security state.

This bit is banked between Security states.

The possible values of this bit are:

**0**  
MemManage exception not pending for the selected Security state.

**1**  
MemManage exception pending for the selected Security state.

If the Main Extension is not implemented, this bit is **RES0**.

This bit resets to zero on a Warm reset.

**USGFAULTPENDED, bit [12]**

UsageFault exception pended state. The UsageFault exception is banked between Security states, this bit indicates and allows modification of the pending state of the UsageFault exception for the selected Security state.

This bit is banked between Security states.

The possible values of this bit are:

**0**  
UsageFault exception not pending for the selected Security state.

**1**  
UsageFault exception pending for the selected Security state.

If the Main Extension is not implemented, this bit is **RES0**.

This bit resets to zero on a Warm reset.

**SYSTICKACT, bit [11]**

SysTick exception active state. This bit indicates and allows modification of the active state of the SysTick exception for the selected Security state.

If two SysTick timers are implemented this bit is banked between Security states.

If less than two SysTick timers are implemented this bit is not banked between Security states.

The possible values of this bit are:

**0**

SysTick exception not active for the selected Security state.

**1**

SysTick exception active for the selected Security state.

If two timers are implemented, then SYSTICKACT is banked between Security states. If one timer is implemented this bit corresponds to the Secure state if ICSR.STTNS is zero, or the Non-secure state if ICSR.STTNS is one.

This bit resets to zero on a Warm reset.

**PENDSVACT, bit [10]**

PendSV exception active state. This bit indicates and allows modification of the active state of the PendSV exception for the selected Security state.

This bit is banked between Security states.

The possible values of this bit are:

**0**

PendSV exception not active for the selected Security state.

**1**

PendSV exception active for the selected Security state.

This bit resets to zero on a Warm reset.

**Bit [9]**

Reserved, [RES0](#).

**MONITORACT, bit [8]**

DebugMonitor exception active state. This bit indicates and allows modification of the active state of the DebugMonitor exception.

This bit is not banked between Security states.

The possible values of this bit are:

**0**

DebugMonitor exception not active.

**1**

DebugMonitor exception active.

If DEMCR.SDME is one this bit is RAZ/WI from Non-secure state

If the Main Extension is not implemented, this bit is [RES0](#).

This bit resets to zero on a Warm reset.

**SVCALLACT, bit [7]**

SVCALL exception active state. This bit indicates and allows modification of the active state of the SVCALL exception for the selected Security state.

This bit is banked between Security states.

The possible values of this bit are:

**0**  
SVCALL exception not active for the selected Security state.

**1**  
SVCALL exception active for the selected Security state.

This bit resets to zero on a Warm reset.

**Bit [6]**

Reserved, [RES0](#).

**NMIACT, bit [5]**

NMI exception active state. This bit indicates and allows modification of the active state of the NMI exception.

This bit is not banked between Security states.

The possible values of this bit are:

**0**  
NMI exception not active.

**1**  
NMI exception active.

The NMI exception is not banked between Security states. When AIRCR.BFHFNMINs is zero, the Non-secure view of this bit is RAZ/WI. This field ignores writes if either the value being written is one, AIRCR.BFHFNMINs is zero, the access is from Non-secure state, the access is not via the NS alias, or the access is from a debugger when DHCSR.S\_SDE is zero. This bit can only be cleared by access from the Secure state to the NS alias.

This bit resets to zero on a Warm reset.

**SECUREFAULTACT, bit [4]**

SecureFault exception active state. This bit indicates and allows modification of the active state of the SecureFault exception.

This bit is not banked between Security states.

The possible values of this bit are:

**0**  
SecureFault exception not active.

**1**  
SecureFault exception active.

This bit is RAZ/WI from Non-secure state.

This bit resets to zero on a Warm reset.

**USGFAULTACT, bit [3]**

UsageFault exception active state for the selected Security state. This bit indicates and allows modification of the active state of the UsageFault exception for the selected Security state.

This bit is banked between Security states.

The possible values of this bit are:

**0**  
UsageFault exception not active for the selected Security state.

**1**  
UsageFault exception active for the selected Security state.

If the Main Extension is not implemented, this bit is [RES0](#).

This bit resets to zero on a Warm reset.

**HARDFULTACT, bit [2]**

HardFault exception active state. Indicates and allows limited modification of the active state of the HardFault exception for the selected Security state.

This bit is banked between Security states.

The possible values of this bit are:

**0**

HardFault exception not active for the selected Security state.

**1**

HardFault exception active for the selected Security state.

This field ignores writes if either the value being written is one, the write targets the Secure HardFault active bit, the access is from Non-secure state, or the access is from a debugger when DHCSR.S\_SDE is zero.

This bit resets to zero on a Warm reset.

**BUSEFAULTACT, bit [1]**

BusFault exception active state. This bit indicates and allows modification of the active state of the BusFault exception.

This bit is not banked between Security states.

The possible values of this bit are:

**0**

BusFault exception not active.

**1**

BusFault exception active.

The BusFault exception is not banked between Security states.

If AIRCR.BFHFNMINS is zero this bit is RAZ/WI from Non-secure state.

If the Main Extension is not implemented, this bit is [RES0](#).

This bit resets to zero on a Warm reset.

**MEMFAULTACT, bit [0]**

MemManage exception active state for the selected Security state. This bit indicates and allows modification of the active state of the MemManage exception for the selected Security state.

This bit is banked between Security states.

The possible values of this bit are:

**0**

MemManage exception not active for the selected Security state.

**1**

MemManage exception active for the selected Security state.

If the Main Extension is not implemented, this bit is [RES0](#).

This bit resets to zero on a Warm reset.

## D1.2.182 SHPR1, System Handler Priority Register 1

The SHPR1 characteristics are:

### Purpose

Sets or returns priority for system handlers 4 - 7.

### Usage constraints

Privileged access permitted only. Unprivileged accesses generate a fault.

If the Main Extension is implemented, this register is word, halfword, and byte accessible.

If the Main Extension is not implemented, this register is word accessible only, halfword and byte accesses are **UNPREDICTABLE**.

### Configurations

Present only if the Main Extension is implemented.

This register is **RES0** if the Main Extension is not implemented.

### Attributes

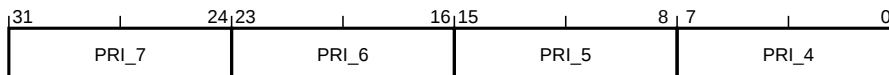
32-bit read/write register located at 0xE000ED18.

Secure software can access the Non-secure version of this register via SHPR1\_NS located at 0xE002ED18. The location 0xE002ED18 is **RES0** to software executing in Non-secure state and the debugger.

This register is banked between Security states on a bit by bit basis.

## Field descriptions

The SHPR1 bit assignments are:



### PRI\_7, bits [31:24]

Priority 7. Priority of system handler 7, SecureFault.

This field is not banked between Security states.

If the PE implements fewer than 8 bits of priority, then the least significant bits of this field are **RES0**.

This field is RAZ/WI from Non-secure state.

This field resets to zero on a Warm reset.

### PRI\_6, bits [23:16]

Priority 6. Priority of system handler 6, UsageFault.

This field is banked between Security states.

If the PE implements fewer than 8 bits of priority, then the least significant bits of this field are **RES0**.

This field resets to zero on a Warm reset.

### PRI\_5, bits [15:8]

Priority 5. Priority of system handler 5, BusFault.

This field is not banked between Security states.

If the PE implements fewer than 8 bits of priority, then the least significant bits of this field are **RES0**.

If AIRCR.BFHFNMINS is zero this field is RAZ/WI from Non-secure state.

This field resets to zero on a Warm reset.

**PRI\_4, bits [7:0]**

Priority 4. Priority of system handler 4, MemManage.

This field is banked between Security states.

If the PE implements fewer than 8 bits of priority, then the least significant bits of this field are [RES0](#).

This field resets to zero on a Warm reset.



## D1.2.183 SHPR2, System Handler Priority Register 2

The SHPR2 characteristics are:

### Purpose

Sets or returns priority for system handlers 8 - 11.

### Usage constraints

Privileged access permitted only. Unprivileged accesses generate a fault.

If the Main Extension is implemented, this register is word, halfword, and byte accessible.

If the Main Extension is not implemented, this register is word accessible only, halfword and byte accesses are **UNPREDICTABLE**.

### Configurations

This register is always implemented.

### Attributes

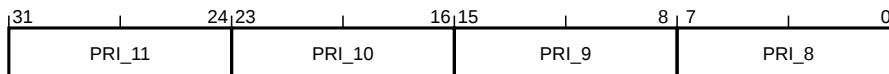
32-bit read/write register located at 0xE000ED1C.

Secure software can access the Non-secure version of this register via SHPR2\_NS located at 0xE002ED1C. The location 0xE002ED1C is **RES0** to software executing in Non-secure state and the debugger.

This register is banked between Security states.

## Field descriptions

The SHPR2 bit assignments are:



### PRI\_11, bits [31:24]

Priority 11. Priority of system handler 11, SVCcall.

This field is banked between Security states.

If the PE implements fewer than 8 bits of priority, then the least significant bits of this field are **RES0**.

This field resets to zero on a Warm reset.

### PRI\_10, bits [23:16]

Reserved, **RES0**.

### PRI\_9, bits [15:8]

Reserved, **RES0**.

### PRI\_8, bits [7:0]

Reserved, **RES0**.

## D1.2.184 SHPR3, System Handler Priority Register 3

The SHPR3 characteristics are:

### Purpose

Sets or returns priority for system handlers 12 - 15.

### Usage constraints

Privileged access permitted only. Unprivileged accesses generate a fault.

If the Main Extension is implemented, this register is word, halfword, and byte accessible.

If the Main Extension is not implemented, this register is word accessible only, halfword and byte accesses are **UNPREDICTABLE**.

### Configurations

This register is always implemented.

### Attributes

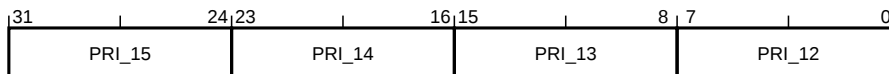
32-bit read/write register located at 0xE000ED20.

Secure software can access the Non-secure version of this register via SHPR3\_NS located at 0xE002ED20. The location 0xE002ED20 is **RES0** to software executing in Non-secure state and the debugger.

This register is banked between Security states on a bit by bit basis.

## Field descriptions

The SHPR3 bit assignments are:



### PRI\_15, bits [31:24]

Priority 15. Priority of system handler 15, SysTick.

If two SysTick timers are implemented this field is banked between Security states.

If less than two SysTick timers are implemented this field is not banked between Security states.

If the PE implements fewer than 8 bits of priority, then the least significant bits of this field are **RES0**. If one timer is implemented, this field corresponds to the Secure state if ICSR.STTNS is zero, or the Non-secure state if ICSR.STTNS is one.

If only one SysTick timer is implemented and ICSR.STTNS is clear, this field is RAZ/WI from Non-secure state.

If no SysTick timer is implemented this field is **RES0**.

This field resets to zero on a Warm reset.

### PRI\_14, bits [23:16]

Priority 14. Priority of system handler 14, PendSV.

This field is banked between Security states.

If the PE implements fewer than 8 bits of priority, then the least significant bits of this field are **RES0**.

This field resets to zero on a Warm reset.

### PRI\_13, bits [15:8]

Reserved, **RES0**.

**PRI\_12, bits [7:0]**

Priority 12. Priority of system handler 12, DebugMonitor.

This field is not banked between Security states.

If the PE implements fewer than 8 bits of priority, then the least significant bits of this field are [RES0](#).

If DEMCR.SDME is one this field is RAZ/WI from Non-secure state

If the Main Extension is not implemented, this field is [RES0](#).

This field resets to zero on a Warm reset.

### D1.2.185 SP, Current Stack Pointer Register

The SP characteristics are:

**Purpose**

Exception and procedure stack pointer register.

**Usage constraints**

Privileged and unprivileged access permitted.

**Configurations**

This register is always implemented.

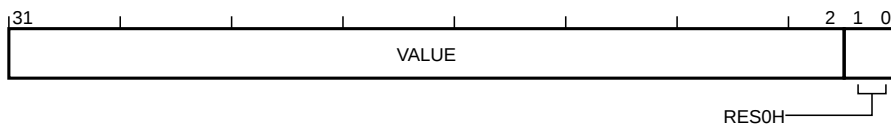
**Attributes**

32-bit read/write special-purpose register.

This register is not banked between Security states.

### Field descriptions

The SP bit assignments are:



**VALUE, bits [31:2]**

Stack pointer. Holds bits[31:2] of the stack pointer address. The current stack pointer is selected from one of MSP\_NS, PSP\_NS, MSP\_S or PSP\_S.

Software can refer to SP as R13.

**Bits [1:0]**

Reserved, [RES0H](#).

### D1.2.186 SP\_NS, Stack Pointer (Non-secure)

The SP\_NS characteristics are:

**Purpose**

Provides access to the current Non-secure stack pointer.

**Usage constraints**

Privileged access only. Unprivileged access is RAZ/WI, unless otherwise stated.

**Configurations**

This register is always implemented.

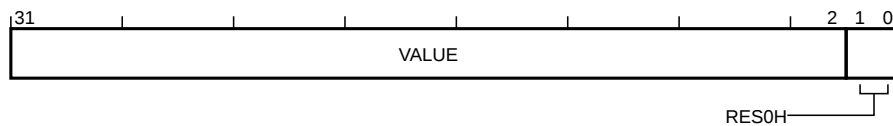
**Attributes**

32-bit read/write special-purpose register.

This register is not banked between Security states.

### Field descriptions

The SP\_NS bit assignments are:



**VALUE, bits [31:2]**

Stack pointer. Holds bits[31:2] of the current Non-secure stack pointer address. SP\_NS is selected from one of MSP\_NS or PSP\_NS. Access to SP\_NS is provided via MRS and MSR and is subject to stack limit checking.

**Bits [1:0]**

Reserved, **RES0H**.



## D1.2.188 SYST\_CALIB, SysTick Calibration Value Register

The SYST\_CALIB characteristics are:

### Purpose

Reads the SysTick timer calibration value and parameters for the selected Security state.

### Usage constraints

Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are **UNPREDICTABLE**.

### Configurations

Present only if at least one SysTick timer is implemented.

This register is **RES0** if no SysTick timer is implemented.

### Attributes

32-bit read-only register located at 0xE000E01C.

Secure software can access the Non-secure version of this register via SYST\_CALIB\_NS located at 0xE002E01C. The location 0xE002E01C is **RES0** to software executing in Non-secure state and the debugger.

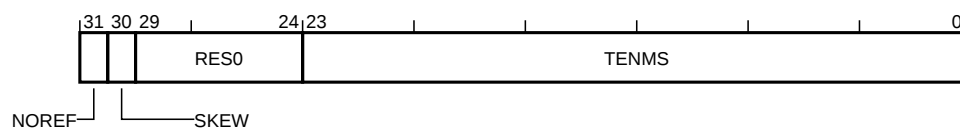
This register is banked between Security states.

### Preface

If the Main Extension is implemented then, two SysTick timers are implemented. If the Main Extension is not implemented, then it is **IMPLEMENTATION DEFINED** whether none, one or two SysTick timers are implemented. Where two SysTick timers are implemented, this register is banked. Where one SysTick timer is implemented, this register is not banked, and Non-secure accesses behave as RAZ/WI if ICSR.STTNS is clear. If no SysTick timer is implemented, both aliases of this register behave as **RES0**.

### Field descriptions

The SYST\_CALIB bit assignments are:



### NOREF, bit [31]

No reference. Indicates whether the **IMPLEMENTATION DEFINED** reference clock is implemented.

The possible values of this bit are:

**0**

Reference clock is implemented.

**1**

Reference clock is not implemented.

When this bit is 1, the CLKSOURCE bit of the SYST\_CSR register is forced to 1 and cannot be cleared to 0.

If only one SysTick timer is implemented and ICSR.STTNS is clear, this bit is RAZ/WI from Non-secure state.

If no SysTick timer is implemented this bit is **RES0**.

This bit resets to an **IMPLEMENTATION DEFINED** value on a Warm reset.

This bit reads as an **IMPLEMENTATION DEFINED** value.

**SKEW, bit [30]**

Skew. Indicates whether the 10ms calibration value is exact.

The possible values of this bit are:

**0**

TENMS calibration value is exact.

**1**

TENMS calibration value is inexact.

If only one SysTick timer is implemented and ICSR.STTNS is clear, this bit is RAZ/WI from Non-secure state.

If no SysTick timer is implemented this bit is **RES0**.

This bit resets to an **IMPLEMENTATION DEFINED** value on a Warm reset.

This bit reads as an **IMPLEMENTATION DEFINED** value.

**Bits [29:24]**

Reserved, **RES0**.

**TENMS, bits [23:0]**

Ten milliseconds. Optionally holds a reload value to be used for 10ms (100Hz) timing, subject to system clock skew errors. If this field is zero, the calibration value is not known.

If only one SysTick timer is implemented and ICSR.STTNS is clear, this field is RAZ/WI from Non-secure state.

If no SysTick timer is implemented this field is **RES0**.

This field resets to an **IMPLEMENTATION DEFINED** value on a Warm reset.

This field reads as an **IMPLEMENTATION DEFINED** value.



## D1.2.189 SYST\_CSR, SysTick Control and Status Register

The SYST\_CSR characteristics are:

### Purpose

Controls the SysTick timer and provides status data for the selected Security state.

### Usage constraints

Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are **UNPREDICTABLE**.

### Configurations

Present only if at least one SysTick timer is implemented.

This register is **RES0** if no SysTick timer is implemented.

### Attributes

32-bit read/write register located at 0xE000E010.

Secure software can access the Non-secure version of this register via SYST\_CSR\_NS located at 0xE002E010. The location 0xE002E010 is **RES0** to software executing in Non-secure state and the debugger.

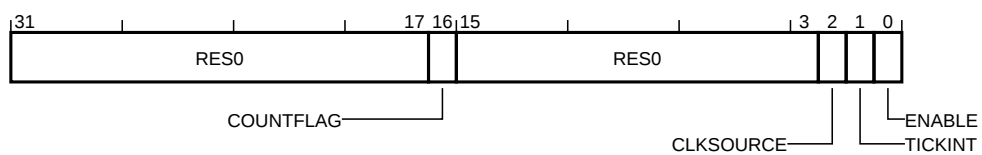
This register is banked between Security states.

### Preface

If the Main Extension is implemented, then two SysTick timers are implemented. If the Main Extension is not implemented, then it is **IMPLEMENTATION DEFINED** whether none, one or two SysTick timers are implemented. Where two SysTick timers are implemented, this register is banked. Where one SysTick timer is implemented, this register is not banked, and Non-secure accesses behave as RAZ/WI if ICSR.STTNS is clear. If no SysTick timer is implemented, both aliases of this register behave as **RES0**.

### Field descriptions

The SYST\_CSR bit assignments are:



#### Bits [31:17]

Reserved, **RES0**.

#### COUNTFLAG, bit [16]

Count flag. Indicates whether the counter has counted to zero since the last read of this register.

The possible values of this bit are:

- 0**  
Timer has not counted to 0.
- 1**  
Timer has counted to 0.

COUNTFLAG is set to 1 by a count transition from 1 to 0. COUNTFLAG is cleared to 0 if software reads this bit as one, and by any write to the SYST\_CVR for the selected Security state. Debugger reads do not clear the COUNTFLAG.

If set this bit clears to zero when read by software. Reads from the debugger do not clear this bit.

If only one SysTick timer is implemented and ICSR.STTNS is clear, this bit is RAZ/WI from Non-secure state.

If no SysTick timer is implemented this bit is [RES0](#).

This bit resets to zero on a Warm reset.

**Bits [15:3]**

Reserved, [RES0](#).

**CLKSOURCE, bit [2]**

Clock source. Indicates the SysTick clock source.

The possible values of this bit are:

**0**

Uses the [IMPLEMENTATION DEFINED](#) external reference clock.

**1**

Uses the PE clock.

If no external clock is implemented, this bit reads as 1 and ignores writes.

If only one SysTick timer is implemented and ICSR.STTNS is clear, this bit is RAZ/WI from Non-secure state.

If no SysTick timer is implemented this bit is [RES0](#).

This bit resets to an [IMPLEMENTATION DEFINED](#) value on a Warm reset.

**TICKINT, bit [1]**

Tick interrupt. Indicates whether counting to 0 causes the status of the SysTick exception to change to pending.

The possible values of this bit are:

**0**

Count to 0 does not affect the SysTick exception status.

**1**

Count to 0 changes the SysTick exception status to pending.

Changing the value of the counter to 0 by writing the SysTick does not change the status of the SysTick exception.

If only one SysTick timer is implemented and ICSR.STTNS is clear, this bit is RAZ/WI from Non-secure state.

If no SysTick timer is implemented this bit is [RES0](#).

This bit resets to zero on a Warm reset.

**ENABLE, bit [0]**

SysTick enable. Indicates the enabled status of the SysTick counter.

The possible values of this bit are:

**0**

Counter is disabled.

**1**

Counter is enabled.

If only one SysTick timer is implemented and ICSR.STTNS is clear, this bit is RAZ/WI from Non-secure state.

If no SysTick timer is implemented this bit is [RES0](#).

This bit resets to zero on a Warm reset.

## D1.2.190 SYST\_CVR, SysTick Current Value Register

The SYST\_CVR characteristics are:

### Purpose

Reads or clears the SysTick timer current counter value for the selected Security state.

### Usage constraints

Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are **UNPREDICTABLE**.

### Configurations

Present only if at least one SysTick timer is implemented.

This register is **RES0** if no SysTick timer is implemented.

### Attributes

32-bit read/write-to-clear register located at 0xE000E018.

Secure software can access the Non-secure version of this register via SYST\_CVR\_NS located at 0xE002E018. The location 0xE002E018 is **RES0** to software executing in Non-secure state and the debugger.

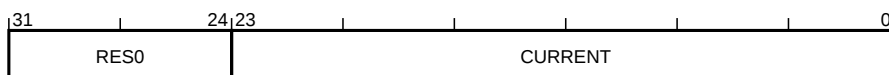
This register is banked between Security states.

## Preface

If the Main Extension is implemented, then two SysTick timers are implemented. If the Main Extension is not implemented, then it is **IMPLEMENTATION DEFINED** whether none, one or two SysTick timers are implemented. Where two SysTick timers are implemented, this register is banked. Where one SysTick timer is implemented, this register is not banked, and Non-secure accesses behave as RAZ/WI if ICSR.STTNS is clear. If no SysTick timer is implemented, both aliases of this register behave as **RES0**.

## Field descriptions

The SYST\_CVR bit assignments are:



### Bits [31:24]

Reserved, **RES0**.

### CURRENT, bits [23:0], on a read

Current counter value. Provides the value of the SysTick timer counter for the selected Security state.

It is **IMPLEMENTATION DEFINED** whether the current counter value decrements if the PE is sleeping and SCR.SLEEPDEEP is set.

If only one SysTick timer is implemented and ICSR.STTNS is clear, this field is RAZ/WI from Non-secure state.

If no SysTick timer is implemented this field is **RES0**.

This field resets to an **UNKNOWN** value on a Warm reset.

### CURRENT, bits [23:0], on a write

Reset counter value. Writing any value clears the SysTick timer counter for the selected Security state to zero.

## D1.2.191 SYST\_RVR, SysTick Reload Value Register

The SYST\_RVR characteristics are:

### Purpose

Provides access SysTick timer counter reload value for the selected Security state.

### Usage constraints

Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are **UNPREDICTABLE**.

### Configurations

Present only if at least one SysTick timer is implemented.

This register is **RES0** if no SysTick timer is implemented.

### Attributes

32-bit read/write register located at 0xE000E014.

Secure software can access the Non-secure version of this register via SYST\_RVR\_NS located at 0xE002E014. The location 0xE002E014 is **RES0** to software executing in Non-secure state and the debugger.

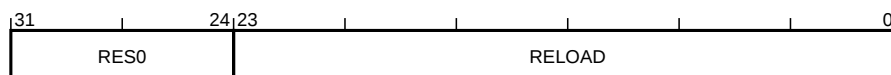
This register is banked between Security states.

## Preface

If the Main Extension is implemented, then two SysTick timers are implemented. If the Main Extension is not implemented, then it is **IMPLEMENTATION DEFINED** whether none, one or two SysTick timers are implemented. Where two SysTick timers are implemented, this register is banked. Where one SysTick timer is implemented, this register is not banked, and Non-secure accesses behave as RAZ/WI if ICSR.STTNS is clear. If no SysTick timer is implemented, both instances of this register behave as **RES0**.

## Field descriptions

The SYST\_RVR bit assignments are:



### Bits [31:24]

Reserved, **RES0**.

### RELOAD, bits [23:0]

Counter reload value. The value to load into the SYST\_CVR for the selected Security state when the counter reaches 0.

If only one SysTick timer is implemented and ICSR.STTNS is clear, this field is RAZ/WI from Non-secure state.

If no SysTick timer is implemented this field is **RES0**.

This field resets to an **UNKNOWN** value on a Warm reset.



### D1.2.193 TPIU\_CIDR0, TPIU Component Identification Register 0

The TPIU\_CIDR0 characteristics are:

#### Purpose

Provides CoreSight Unique Component Identifier information for the TPIU.

#### Usage constraints

Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are **UNPREDICTABLE**.

If the Main Extension is not implemented then it is **IMPLEMENTATION DEFINED** whether this register is accessible only to the debugger and **RES0** for software. Otherwise the register is accessible to the debugger and software.

#### Configurations

Present only if CoreSight identification is implemented.

This register is **RES0** if CoreSight identification is not implemented.

Present only if the TPIU is implemented.

This register is **RES0** if the TPIU is not implemented.

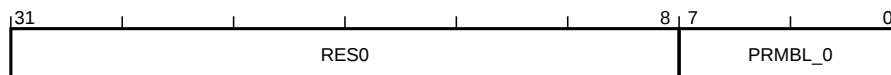
#### Attributes

32-bit read-only register located at 0xE0040FF0.

This register is not banked between Security states.

### Field descriptions

The TPIU\_CIDR0 bit assignments are:



#### Bits [31:8]

Reserved, **RES0**.

#### PRMBL\_0, bits [7:0]

CoreSight component identification preamble. See the *Arm® CoreSight™ Architecture Specification*.

This field reads as 0x0D.

## D1.2.194 TPIU\_CIDR1, TPIU Component Identification Register 1

The TPIU\_CIDR1 characteristics are:

### Purpose

Provides CoreSight Unique Component Identifier information for the TPIU.

### Usage constraints

Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are **UNPREDICTABLE**.

If the Main Extension is not implemented then it is **IMPLEMENTATION DEFINED** whether this register is accessible only to the debugger and **RES0** for software. Otherwise the register is accessible to the debugger and software.

### Configurations

Present only if CoreSight identification is implemented.

This register is **RES0** if CoreSight identification is not implemented.

Present only if the TPIU is implemented.

This register is **RES0** if the TPIU is not implemented.

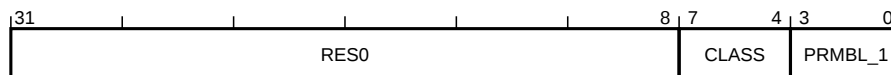
### Attributes

32-bit read-only register located at 0xE0040FF4.

This register is not banked between Security states.

## Field descriptions

The TPIU\_CIDR1 bit assignments are:



### Bits [31:8]

Reserved, **RES0**.

### CLASS, bits [7:4]

CoreSight component class. See the *Arm® CoreSight™ Architecture Specification*.

This field reads as 0x9.

### PRMBL\_1, bits [3:0]

CoreSight component identification preamble. See the *Arm® CoreSight™ Architecture Specification*.

This field reads as 0x0.



## D1.2.195 TPIU\_CIDR2, TPIU Component Identification Register 2

The TPIU\_CIDR2 characteristics are:

### Purpose

Provides CoreSight Unique Component Identifier information for the TPIU.

### Usage constraints

Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are **UNPREDICTABLE**.

If the Main Extension is not implemented then it is **IMPLEMENTATION DEFINED** whether this register is accessible only to the debugger and **RES0** for software. Otherwise the register is accessible to the debugger and software.

### Configurations

Present only if CoreSight identification is implemented.

This register is **RES0** if CoreSight identification is not implemented.

Present only if the TPIU is implemented.

This register is **RES0** if the TPIU is not implemented.

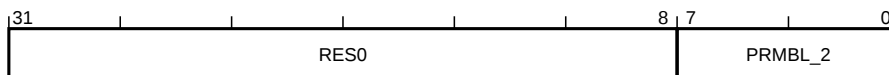
### Attributes

32-bit read-only register located at 0xE0040FF8.

This register is not banked between Security states.

## Field descriptions

The TPIU\_CIDR2 bit assignments are:



### Bits [31:8]

Reserved, **RES0**.

### PRMBL\_2, bits [7:0]

CoreSight component identification preamble. See the *Arm® CoreSight™ Architecture Specification*.

This field reads as 0x05.



## D1.2.197 TPIU\_CSPSR, TPIU Current Parallel Port Sizes Register

The TPIU\_CSPSR characteristics are:

### Purpose

Controls the width of the parallel trace port.

### Usage constraints

Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are **UNPREDICTABLE**.

If the Main Extension is not implemented then it is **IMPLEMENTATION DEFINED** whether this register is accessible only to the debugger and **RES0** for software. Otherwise the register is accessible to the debugger and software.

### Configurations

Present only if the TPIU is implemented.

This register is **RES0** if the TPIU is not implemented.

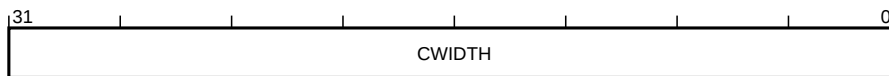
### Attributes

32-bit read/write register located at 0xE0040004.

This register is not banked between Security states.

## Field descriptions

The TPIU\_CSPSR bit assignments are:



### CWIDTH, bits [31:0]

Current width. CWIDTH[*m*] represents a parallel trace port width of (*m*+1).

The possible values of each bit are:

**0**

Width (N+1) is not the current parallel trace port width.

**1**

Width (N+1) is the current parallel trace port width.

A debugger must set only one bit is set to 1, and all others must be zero. The effect of writing a value with more than one bit set to 1 is **UNPREDICTABLE**. The effect of a write to an unsupported bit is **UNPREDICTABLE**.

This register resets to the value for the smallest supported parallel trace port size.

This field resets to an **IMPLEMENTATION DEFINED** value on a Cold reset.

## D1.2.198 TPIU\_DEVTYPE, TPIU Device Type Register

The TPIU\_DEVTYPE characteristics are:

### Purpose

Provides CoreSight Unique Component Identifier information for the TPIU.

### Usage constraints

Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

If the Main Extension is not implemented then it is IMPLEMENTATION DEFINED whether this register is accessible only to the debugger and RES0 for software. Otherwise the register is accessible to the debugger and software.

### Configurations

Present only if CoreSight identification is implemented.

This register is RES0 if CoreSight identification is not implemented.

Present only if the TPIU is implemented.

This register is RES0 if the TPIU is not implemented.

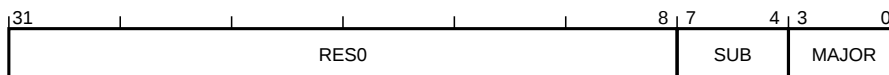
### Attributes

32-bit read-only register located at 0xE0040FCC.

This register is not banked between Security states.

## Field descriptions

The TPIU\_DEVTYPE bit assignments are:



### Bits [31:8]

Reserved, RES0.

### SUB, bits [7:4]

Sub-type. Component sub-type.

The possible values of this field are:

#### 0x0

Other. Only permitted if the MAJOR field reads as 0x0.

#### 0x1

Trace port. Only permitted if the MAJOR field reads as 0x1.

This field reads as an IMPLEMENTATION DEFINED value.

### MAJOR, bits [3:0]

Major type. Component major type.

The possible values of this field are:

#### 0x0

Miscellaneous.

#### 0x1

Trace sink.

This field reads as an **IMPLEMENTATION DEFINED** value.

## D1.2.199 TPIU\_FFCR, TPIU Formatter and Flush Control Register

The TPIU\_FFCR characteristics are:

### Purpose

Controls the TPIU formatter. This register might contain other formatter and flush control fields that are outside the scope of the architecture. Contact Arm for more information.

### Usage constraints

Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are **UNPREDICTABLE**.

If the Main Extension is not implemented then it is **IMPLEMENTATION DEFINED** whether this register is accessible only to the debugger and **RES0** for software. Otherwise the register is accessible to the debugger and software.

### Configurations

Present only if the TPIU is implemented.

This register is **RES0** if the TPIU is not implemented.

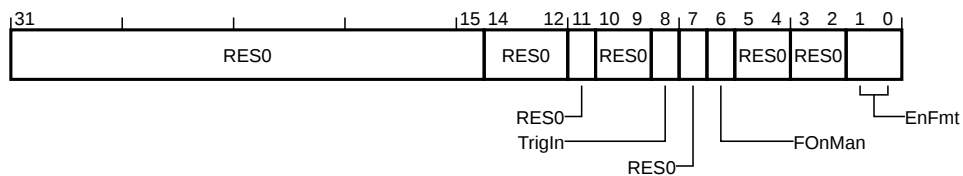
### Attributes

32-bit read/write register located at 0xE0040304.

This register is not banked between Security states.

## Field descriptions

The TPIU\_FFCR bit assignments are:



### Bits [31:15,11,7,3:2]

Reserved, **RES0**.

### Bits [14:12]

Reserved for formatter stop controls.

Reserved, **RES0**.

### Bits [10:9]

Reserved for additional trigger mark controls.

Reserved, **RES0**.

### TrigIn, bit [8]

Trigger input asserted. Indicate a trigger on the trace port when an **IMPLEMENTATION DEFINED** TRIGIN signal is asserted.

It is **IMPLEMENTATION DEFINED** whether this bit is R/W or RAO.

This bit resets to zero on a Cold reset.

### FOnMan, bit [6]

Flush On Manual. Setting this bit to 1 generates a flush. The TPIU clears the bit to 0 when the flush completes.

This bit resets to zero on a Cold reset.

**Bits [5:4]**

Reserved for additional flush controls.

Reserved, [RES0](#).

**EnFmt, bits [1:0]**

Formatter control. Selects the output formatting mode.

The possible values of this field are:

**0b00**

Bypass. Disable formatting. Only supported when SWO mode is selected. Only a single trace source is supported in bypass mode:

- If only a single trace source is connected to this TPIU, it is selected.
- If multiple sources (including the ITM) are implemented and connected to this TPIU, then all other trace sources, except for the ITM, must be disabled. Otherwise, the trace output is [UNPREDICTABLE](#).

All other trace sources are discarded.

**0b10**

Continuous. Enable formatting and embed triggers and null cycles in the formatted output.

All other values are reserved.

If no formatter is implemented, this field is [RES0](#). This field must be set to 0b10 when the parallel trace port is selected, or when using multiple trace sources. Changing the value of this field when TPIU\_FFSR.FtStopped is 0 is [UNPREDICTABLE](#).

This field resets to zero on a Cold reset.

**Note**

An optional TRACECTL pin might be implemented as part of the parallel trace port that allows Bypass mode when using a parallel trace port and a further mode, EnFmt == 0b01. The CoreSight architecture describes EnFmt[1] as the EnFCont bit and EnFmt[0] as the EnFTC bit.

## D1.2.200 TPIU\_FFSR, TPIU Formatter and Flush Status Register

The TPIU\_FFSR characteristics are:

### Purpose

Shows the status and capabilities of the TPIU formatter.

### Usage constraints

Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are **UNPREDICTABLE**.

If the Main Extension is not implemented then it is **IMPLEMENTATION DEFINED** whether this register is accessible only to the debugger and **RES0** for software. Otherwise the register is accessible to the debugger and software.

### Configurations

Present only if the TPIU is implemented.

This register is **RES0** if the TPIU is not implemented.

### Attributes

32-bit read-only register located at 0xE0040300.

This register is not banked between Security states.

## Field descriptions

The TPIU\_FFSR bit assignments are:



### Bits [31:4]

Reserved, **RES0**.

### FtNonStop, bit [3]

Non-stop formatter. Indicates the formatter cannot be stopped.

The possible values of this bit are:

**0**

Formatter can be stopped.

**1**

Formatter cannot be stopped.

If no formatter is implemented, this bit is RAO.

### TCPresent, bit [2]

TRACECTL present. Indicates presence of the TRACECTL pin.

The possible values of this bit are:

**0**

No TRACECTL pin is available. The data formatter must be used and only in continuous mode.

**1**

The optional TRACECTL pin is present.



If a parallel trace port is not implemented, this bit is RAZ.

**Note**

If a parallel trace port is implemented, Arm recommends the TRACECTL pin is not implemented.

**FtStopped, bit [1]**

Formatter stopped. Indicates the formatter is stopped.

The possible values of this bit are:

**0**

Formatter is enabled.

**1**

The formatter has received a stop request signal and all trace data and post-amble has been output. Any further trace data is ignored.

If no formatter is implemented, or the formatter cannot be stopped, this bit is RAZ.

**FInProg, bit [0]**

Flush in progress. Set to 1 when a flush is initiated and clears to zero when all data received before the flush is acknowledged has been output on the trace port. That is, the trace has been received at the sink, formatted, and output on the trace port.

### D1.2.201 TPIU\_LAR, TPIU Software Lock Access Register

The TPIU\_LAR characteristics are:

#### Purpose

Provides CoreSight Software Lock control for the TPIU, see the *Arm® CoreSight™ Architecture Specification* for details.

#### Usage constraints

Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are **UNPREDICTABLE**.

This register is RAZ/WI if accessed via the debugger.

#### Configurations

Present only if CoreSight identification is implemented.

This register is **RES0** if CoreSight identification is not implemented.

Present only if the TPIU is implemented.

This register is **RES0** if the TPIU is not implemented.

Present only if the optional Software Lock is implemented.

This register is RAZ/WI if the Software Lock is not implemented.

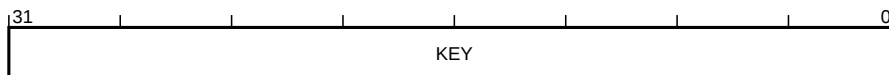
#### Attributes

32-bit write-only register located at 0xE0040FB0.

This register is not banked between Security states.

### Field descriptions

The TPIU\_LAR bit assignments are:



#### KEY, bits [31:0]

Lock Access control.

Writing the key value 0xC5ACCE55 to this field unlocks the lock, enabling write accesses to the registers of this component through a memory-mapped interface.

Writing any other value to this register locks the lock, disabling write accesses to the registers of this component through a memory mapped interface.

## D1.2.202 TPIU\_LSR, TPIU Software Lock Status Register

The TPIU\_LSR characteristics are:

### Purpose

Provides CoreSight Software Lock status information for the TPIU, see the *Arm® CoreSight™ Architecture Specification* for details.

### Usage constraints

Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are **UNPREDICTABLE**.

If the Main Extension is not implemented then it is **IMPLEMENTATION DEFINED** whether this register is accessible only to the debugger and **RES0** for software. Otherwise the register is accessible to the debugger and software.

This register is RAZ/WI if accessed via the debugger.

### Configurations

Present only if CoreSight identification is implemented.

This register is **RES0** if CoreSight identification is not implemented.

Present only if the TPIU is implemented.

This register is **RES0** if the TPIU is not implemented.

Present only if the optional Software Lock is implemented.

This register is RAZ/WI if the Software Lock is not implemented.

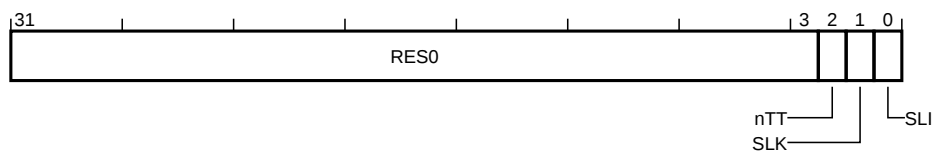
### Attributes

32-bit read-only register located at 0xE0040FB4.

This register is not banked between Security states.

## Field descriptions

The TPIU\_LSR bit assignments are:



### Bits [31:3]

Reserved, **RES0**.

### nTT, bit [2]

Not thirty-two bit. See the *Arm® CoreSight™ Architecture Specification*.

This bit reads as zero.

### SLK, bit [1]

Software Lock status. See the *Arm® CoreSight™ Architecture Specification*.

The possible values of this bit are:

**0**

Lock clear. Software writes are permitted to the registers of this component.

**1**

Lock set. Software writes to the registers of this component are ignored, and reads have no side-effects.

For a debugger read of this register, or when the Software Lock is not implemented, this bit is **RES0**.

This bit resets to one on a Cold reset.

**SLI, bit [0]**

Software Lock implemented. See the *Arm® CoreSight™ Architecture Specification*.

The possible values of this bit are:

**0**

Software Lock not implemented or debugger access.

**1**

Software Lock is implemented and software access.

For a debugger read of this register, or when the Software Lock is not implemented, this bit is **RAZ**.

This bit reads as an **IMPLEMENTATION DEFINED** value.



### D1.2.204 TPIU\_PIDR1, TPIU Peripheral Identification Register 1

The TPIU\_PIDR1 characteristics are:

#### Purpose

Provides CoreSight Unique Component Identifier information for the TPIU.

#### Usage constraints

Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are **UNPREDICTABLE**.

If the Main Extension is not implemented then it is **IMPLEMENTATION DEFINED** whether this register is accessible only to the debugger and **RES0** for software. Otherwise the register is accessible to the debugger and software.

#### Configurations

Present only if CoreSight identification is implemented.

This register is **RES0** if CoreSight identification is not implemented.

Present only if the TPIU is implemented.

This register is **RES0** if the TPIU is not implemented.

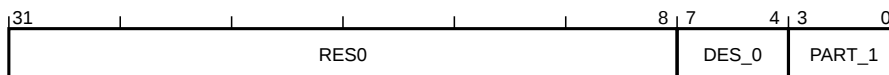
#### Attributes

32-bit read-only register located at 0xE0040FE4.

This register is not banked between Security states.

### Field descriptions

The TPIU\_PIDR1 bit assignments are:



#### Bits [31:8]

Reserved, **RES0**.

#### DES\_0, bits [7:4]

JEP106 identification code bits [3:0]. See the *Arm® CoreSight™ Architecture Specification*.

This field reads as an **IMPLEMENTATION DEFINED** value.

#### PART\_1, bits [3:0]

Part number bits [11:8]. See the *Arm® CoreSight™ Architecture Specification*.

This field reads as an **IMPLEMENTATION DEFINED** value.

## D1.2.205 TPIU\_PIDR2, TPIU Peripheral Identification Register 2

The TPIU\_PIDR2 characteristics are:

### Purpose

Provides CoreSight Unique Component Identifier information for the TPIU.

### Usage constraints

Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are **UNPREDICTABLE**.

If the Main Extension is not implemented then it is **IMPLEMENTATION DEFINED** whether this register is accessible only to the debugger and **RES0** for software. Otherwise the register is accessible to the debugger and software.

### Configurations

Present only if CoreSight identification is implemented.

This register is **RES0** if CoreSight identification is not implemented.

Present only if the TPIU is implemented.

This register is **RES0** if the TPIU is not implemented.

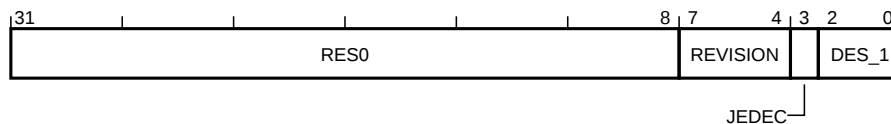
### Attributes

32-bit read-only register located at 0xE0040FE8.

This register is not banked between Security states.

## Field descriptions

The TPIU\_PIDR2 bit assignments are:



### Bits [31:8]

Reserved, **RES0**.

### REVISION, bits [7:4]

Component revision. See the *Arm® CoreSight™ Architecture Specification*.

This field reads as an **IMPLEMENTATION DEFINED** value.

### JEDEC, bit [3]

JEDEC assignee value is used. See the *Arm® CoreSight™ Architecture Specification*.

This bit reads as one.

### DES\_1, bits [2:0]

JEP106 identification code bits [6:4]. See the *Arm® CoreSight™ Architecture Specification*.

This field reads as an **IMPLEMENTATION DEFINED** value.

### D1.2.206 TPIU\_PIDR3, TPIU Peripheral Identification Register 3

The TPIU\_PIDR3 characteristics are:

#### Purpose

Provides CoreSight Unique Component Identifier information for the TPIU.

#### Usage constraints

Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are **UNPREDICTABLE**.

If the Main Extension is not implemented then it is **IMPLEMENTATION DEFINED** whether this register is accessible only to the debugger and **RES0** for software. Otherwise the register is accessible to the debugger and software.

#### Configurations

Present only if CoreSight identification is implemented.

This register is **RES0** if CoreSight identification is not implemented.

Present only if the TPIU is implemented.

This register is **RES0** if the TPIU is not implemented.

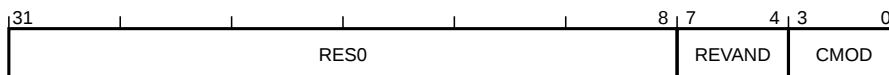
#### Attributes

32-bit read-only register located at 0xE0040FEC.

This register is not banked between Security states.

#### Field descriptions

The TPIU\_PIDR3 bit assignments are:



#### Bits [31:8]

Reserved, **RES0**.

#### REVAND, bits [7:4]

RevAnd. See the *Arm® CoreSight™ Architecture Specification*.

This field reads as an **IMPLEMENTATION DEFINED** value.

#### CMOD, bits [3:0]

Customer Modified. See the *Arm® CoreSight™ Architecture Specification*.

This field reads as an **IMPLEMENTATION DEFINED** value.



## D1.2.207 TPIU\_PIDR4, TPIU Peripheral Identification Register 4

The TPIU\_PIDR4 characteristics are:

### Purpose

Provides CoreSight Unique Component Identifier information for the TPIU.

### Usage constraints

Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are **UNPREDICTABLE**.

If the Main Extension is not implemented then it is **IMPLEMENTATION DEFINED** whether this register is accessible only to the debugger and **RES0** for software. Otherwise the register is accessible to the debugger and software.

### Configurations

Present only if CoreSight identification is implemented.

This register is **RES0** if CoreSight identification is not implemented.

Present only if the TPIU is implemented.

This register is **RES0** if the TPIU is not implemented.

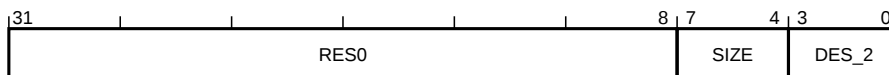
### Attributes

32-bit read-only register located at 0xE0040FD0.

This register is not banked between Security states.

## Field descriptions

The TPIU\_PIDR4 bit assignments are:



### Bits [31:8]

Reserved, **RES0**.

### SIZE, bits [7:4]

4KB count. See the *Arm® CoreSight™ Architecture Specification*.

This field reads as zero.

### DES\_2, bits [3:0]

JEP106 continuation code. See the *Arm® CoreSight™ Architecture Specification*.

This field reads as an **IMPLEMENTATION DEFINED** value.

## D1.2.208 TPIU\_PIDR5, TPIU Peripheral Identification Register 5

The TPIU\_PIDR5 characteristics are:

### Purpose

Provides CoreSight Unique Component Identifier information for the TPIU.

### Usage constraints

Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

If the Main Extension is not implemented then it is IMPLEMENTATION DEFINED whether this register is accessible only to the debugger and RES0 for software. Otherwise the register is accessible to the debugger and software.

### Configurations

Present only if CoreSight identification is implemented.

This register is RES0 if CoreSight identification is not implemented.

Present only if the TPIU is implemented.

This register is RES0 if the TPIU is not implemented.

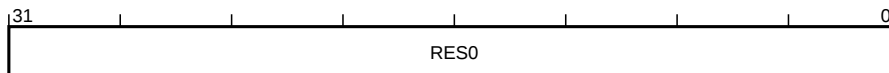
### Attributes

32-bit read-only register located at 0xE0040FD4.

This register is not banked between Security states.

## Field descriptions

The TPIU\_PIDR5 bit assignments are:



### Bits [31:0]

Reserved, RES0.

### D1.2.209 TPIU\_PIDR6, TPIU Peripheral Identification Register 6

The TPIU\_PIDR6 characteristics are:

#### Purpose

Provides CoreSight Unique Component Identifier information for the TPIU.

#### Usage constraints

Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

If the Main Extension is not implemented then it is IMPLEMENTATION DEFINED whether this register is accessible only to the debugger and RES0 for software. Otherwise the register is accessible to the debugger and software.

#### Configurations

Present only if CoreSight identification is implemented.

This register is RES0 if CoreSight identification is not implemented.

Present only if the TPIU is implemented.

This register is RES0 if the TPIU is not implemented.

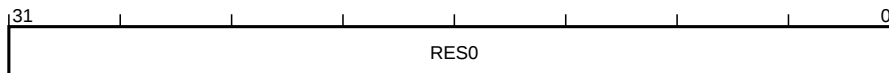
#### Attributes

32-bit read-only register located at 0xE0040FD8.

This register is not banked between Security states.

#### Field descriptions

The TPIU\_PIDR6 bit assignments are:



#### Bits [31:0]

Reserved, RES0.

### D1.2.210 TPIU\_PIDR7, TPIU Peripheral Identification Register 7

The TPIU\_PIDR7 characteristics are:

#### Purpose

Provides CoreSight Unique Component Identifier information for the TPIU.

#### Usage constraints

Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

If the Main Extension is not implemented then it is IMPLEMENTATION DEFINED whether this register is accessible only to the debugger and RES0 for software. Otherwise the register is accessible to the debugger and software.

#### Configurations

Present only if CoreSight identification is implemented.

This register is RES0 if CoreSight identification is not implemented.

Present only if the TPIU is implemented.

This register is RES0 if the TPIU is not implemented.

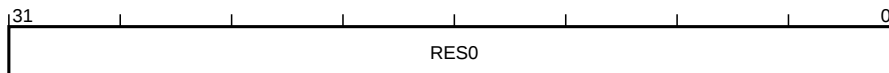
#### Attributes

32-bit read-only register located at 0xE0040FDC.

This register is not banked between Security states.

#### Field descriptions

The TPIU\_PIDR7 bit assignments are:



#### Bits [31:0]

Reserved, RES0.



**0b111111**

$2^{31}$  bytes.

All other values are reserved.

The Periodic Synchronization Counter might have a maximum value smaller than  $2^{31}$ . In this case, if the programmed reload value is greater than the maximum value, then the Periodic Synchronization Counter is reloaded with its maximum value and the TPIU will generate synchronization requests at this interval.

This field resets to 0xA on a Cold reset.

**Note**

In the CoreSight TPIU, TPIU\_PSCR specifies the number of frames between synchronizations, each frame being 16 bytes. This definition of TPIU\_PSCR specifies a number of bytes and is encoded as a power-of-two rather than a plain binary number.

### D1.2.212 TPIU\_SPPR, TPIU Selected Pin Protocol Register

The TPIU\_SPPR characteristics are:

#### Purpose

Selects the protocol used for trace output.

#### Usage constraints

Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are **UNPREDICTABLE**.

If the Main Extension is not implemented then it is **IMPLEMENTATION DEFINED** whether this register is accessible only to the debugger and **RES0** for software. Otherwise the register is accessible to the debugger and software.

If a debugger changes the register value while the TPIU is transmitting data, the effect on the output stream is **UNPREDICTABLE** and the required recovery process is **IMPLEMENTATION DEFINED**.

#### Configurations

Present only if the TPIU is implemented and supports SWO.

This register is **RES0** if the TPIU is not implemented or does not support SWO.

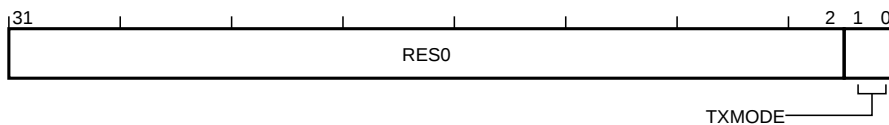
#### Attributes

32-bit read/write register located at 0xE00400F0.

This register is not banked between Security states.

#### Field descriptions

The TPIU\_SPPR bit assignments are:



#### Bits [31:2]

Reserved, **RES0**.

#### TXMODE, bits [1:0]

Transmit mode. Specifies the protocol for trace output from the TPIU.

The possible values of this field are:

##### 0b00

Parallel trace port mode. This value is reserved if TPIU\_TYPE.PTINVALID == 1.

##### 0b01

Asynchronous SWO, using Manchester encoding. This value is reserved if TPIU\_TYPE.MANCVALID == 0.

##### 0b10

Asynchronous SWO, using NRZ encoding. This value is reserved if TPIU\_TYPE.NRZVALID == 0.

All other values are reserved.

The effect of selecting a reserved value, or a mode that the implementation does not support, is **UNPREDICTABLE**.

This field resets to an **IMPLEMENTATION DEFINED** value on a Cold reset.

### D1.2.213 TPIU\_SSPSR, TPIU Supported Parallel Port Sizes Register

The TPIU\_SSPSR characteristics are:

#### Purpose

Indicates the supported parallel trace port sizes.

#### Usage constraints

Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

If the Main Extension is not implemented then it is IMPLEMENTATION DEFINED whether this register is accessible only to the debugger and RES0 for software. Otherwise the register is accessible to the debugger and software.

#### Configurations

Present only if the TPIU is implemented.

This register is RES0 if the TPIU is not implemented.

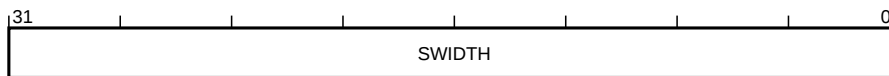
#### Attributes

32-bit read-only register located at 0xE0040000.

This register is not banked between Security states.

### Field descriptions

The TPIU\_SSPSR bit assignments are:



#### SWIDTH, bits [31:0]

Supported width. SWIDTH[ $m$ ] indicates whether a parallel trace port width of  $(m+1)$  is supported.

The possible values of each bit are:

**0**

Parallel trace port width  $(m+1)$  not supported.

**1**

Parallel trace port width  $(m+1)$  supported.

The value of this register is IMPLEMENTATION DEFINED.

This field reads as an IMPLEMENTATION DEFINED value.



### D1.2.214 TPIU\_TYPE, TPIU Device Identifier Register

The TPIU\_TYPE characteristics are:

**Purpose**

Describes the TPIU to a debugger.

**Usage constraints**

Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

If the Main Extension is not implemented then it is IMPLEMENTATION DEFINED whether this register is accessible only to the debugger and RES0 for software. Otherwise the register is accessible to the debugger and software.

**Configurations**

Present only if the TPIU is implemented.

This register is RES0 if the TPIU is not implemented.

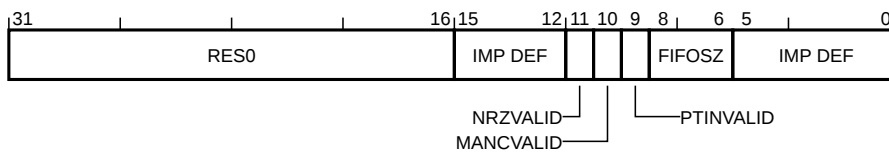
**Attributes**

32-bit read-only register located at 0xE0040FC8.

This register is not banked between Security states.

### Field descriptions

The TPIU\_TYPE bit assignments are:



**Bits [31:16]**

Reserved, RES0.

**Bits [15:12]**

IMPLEMENTATION DEFINED.

**NRZVALID, bit [11]**

NRZ valid. Indicates support for SWO using UART/NRZ encoding.

The possible values of this bit are:

- 0**  
Not supported.
- 1**  
Supported.

This bit reads as an IMPLEMENTATION DEFINED value.

**MANCVALID, bit [10]**

Manchester valid. Indicates support for SWO using Manchester encoding.

The possible values of this bit are:

- 0**  
Not supported.

**1**  
Supported.

This bit reads as an **IMPLEMENTATION DEFINED** value.

**PTINVALID, bit [9]**

Parallel Trace Interface invalid. Indicates support for parallel trace port operation.

The possible values of this bit are:

**0**  
Supported.

**1**  
Not supported.

This bit reads as an **IMPLEMENTATION DEFINED** value.

**FIFOSZ, bits [8:6]**

FIFO depth. Indicates the minimum implemented size of the TPIU output FIFO for trace data.

The possible values of this field are:

**0**  
**IMPLEMENTATION DEFINED** FIFO depth.

**Other**

Minimum FIFO size is  $2^{\text{FIFOSZ}}$ .

For example, a value of `0b011` indicates a FIFO size of at least  $2^3 = 8$  bytes.

This field reads as an **IMPLEMENTATION DEFINED** value.

**Bits [5:0]**

**IMPLEMENTATION DEFINED.**

### D1.2.215 TT\_RESP, Test Target Response Payload

The TT\_RESP characteristics are:

**Purpose**

Provides the response information from a TT, TTA, TTT, or TTAT instruction.

**Usage constraints**

None.

**Configurations**

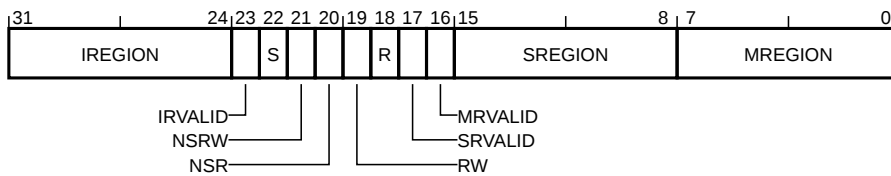
All.

**Attributes**

32-bit payload.

### Field descriptions

The TT\_RESP bit assignments are:



**IREGION, bits [31:24]**

IDAU region number. Indicates the IDAU region number containing the target address.

This field is zero if IRVALID is zero.

**IRVALID, bit [23]**

IREGION valid flag. For a Secure request, indicates the validity of the IREGION field.

The possible values of this bit are:

**0**  
IREGION content not valid.

**1**  
IREGION content valid.

This bit is always zero if the IDAU cannot provide a region number, the address is exempt from security attribution, or if the requesting TT instruction was executed from the Non-secure state.

**S, bit [22]**

Security. For a Secure request, indicates the Security attribute of the target address.

The possible values of this bit are:

**0**  
Target address is Non-secure.

**1**  
Target address is Secure.

This bit is always zero if the requesting TT instruction was executed from the Non-secure state.

**NSRW, bit [21]**

Non-secure read and writable. Equal to RW AND NOT S. Can be used in combination with the LSLS (immediate) instruction to check both the MPU and SAU/IDAU permissions. This field is only valid if the instruction was executed from Secure state and the RW field is valid.

**NSR, bit [20]**

Non-secure readable. Equal to R AND NOT S. Can be used in combination with the LSLS (immediate) instruction to check both the MPU and SAU/IDAU permissions. This field is only valid if the instruction was executed from Secure state and the R field is valid.

**RW, bit [19]**

Read and writable.

Set to 1 if the memory location can be read and written according to the permissions of the selected MPU when operating in the current mode. For TTT and TTAT, this field returns the permissions for unprivileged access, regardless of whether the current mode is privileged or unprivileged.

This field is invalid and RAZ if the TT instruction was executed from an unprivileged mode and the A flag was not specified. This field is also RAZ if the address matches multiple MPU regions.

**R, bit [18]**

Readable.

Read accessibility. Set to 1 if the memory location can be read according to the permissions of the selected MPU when operating in the current mode. For TTT and TTAT, this field returns the permissions for unprivileged access, regardless of whether the current mode is privileged or unprivileged.

This field is invalid and RAZ if the TT instruction was executed from an unprivileged mode and the A flag was not specified. This field is also RAZ if the address matches multiple MPU regions.

**SRVALID, bit [17]**

SREGION valid flag. For a Secure request indicates validity of the SREGION field.

The possible values of this bit are:

**0**

SREGION content not valid.

**1**

SREGION content valid.

This bit is always zero if the requesting TT instruction was executed from the Non-secure state.

The SREGION field is invalid if any of the following are true:

- SAU\_CTRL.ENABLE is set to zero.
- The register argument specified in the SREGION field does not match any enabled SAU regions.
- The address specified matches multiple enabled SAU regions.
- The address specified by the SREGION field is exempt from the secure memory attribution.
- The TT instruction was executed from the Non-secure state or the Security Extension is not implemented.

**MRVALID, bit [16]**

MREGION valid flag. Indicates validity of the MREGION field.

The possible values of this bit are:

**0**

MREGION content not valid.

**1**

MREGION content valid.

This bit is only valid for TT and TTA instructions, executed in the Secure state or in privileged mode in Non-secure state.

The MREGION field is invalid if any of the following is true:

- The MPU is not implemented or MPU\_CTRL.ENABLE is set to zero.

- The register argument specified by the MREGION field does not match any enabled MPU regions.
- The address matched multiple MPU regions.
- The TT instruction was executed from an unprivileged mode and the A flag was not specified.

**SREGION, bits [15:8]**

SAU region number. Holds the SAU region that the address maps to.

This field is only valid if the instruction was executed from Secure state. This field is zero if SRVALID is 0.

**MREGION, bits [7:0]**

MPU region number. Holds the MPU region that the address maps to.

This field is zero if MRVALID is 0.

### D1.2.216 UFSR, UsageFault Status Register

The UFSR characteristics are:

**Purpose**

Contains the status for some instruction execution faults, and for data access faults.

**Usage constraints**

Privileged access permitted only. Unprivileged accesses generate a fault.

If the Main Extension is implemented, this register is word, halfword, and byte accessible.

If the Main Extension is not implemented, this register is word accessible only, halfword and byte accesses are **UNPREDICTABLE**.

**Configurations**

Present only if the Main Extension is implemented.

This register is **RES0** if the Main Extension is not implemented.

**Attributes**

16-bit read/write-one-to-clear register located at 0xE000ED2A.

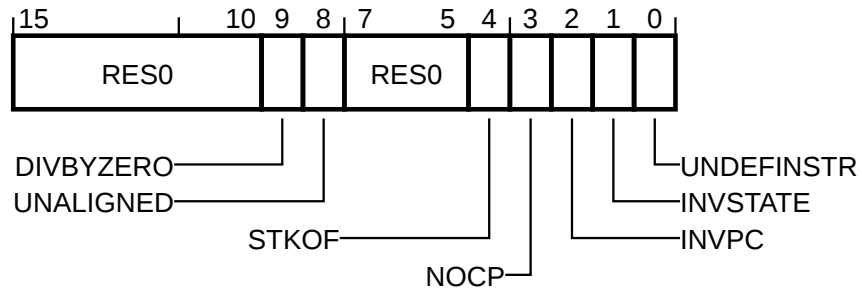
Secure software can access the Non-secure version of this register via UFSR\_NS located at 0xE002ED2A. The location 0xE002ED2A is **RES0** to software executing in Non-secure state and the debugger.

This register is banked between Security states.

This register is part of CFSR.

### Field descriptions

The UFSR bit assignments are:



**Bits [15:10]**

Reserved, **RES0**.

**DIVBYZERO, bit [9]**

Divide by zero flag. Sticky flag indicating whether an integer division by zero error has occurred.

The possible values of this bit are:

**0**  
Error has not occurred.

**1**  
Error has occurred.

This bit resets to zero on a Warm reset.

**UNALIGNED, bit [8]**

Unaligned access flag. Sticky flag indicating whether an unaligned access error has occurred.

The possible values of this bit are:

**0**  
Error has not occurred.

**1**  
Error has occurred.

This bit resets to zero on a Warm reset.

**Bits [7:5]**

Reserved, [RES0](#).

**STKOF, bit [4]**

Stack overflow flag. Sticky flag indicating whether a stack overflow error has occurred.

The possible values of this bit are:

**0**  
Error has not occurred.

**1**  
Error has occurred.

This bit resets to zero on a Warm reset.

**NOCP, bit [3]**

No coprocessor flag. Sticky flag indicating whether a coprocessor disabled or not present error has occurred.

The possible values of this bit are:

**0**  
Error has not occurred.

**1**  
Error has occurred.

This bit resets to zero on a Warm reset.

**INVPC, bit [2]**

Invalid PC flag. Sticky flag indicating whether an integrity check error has occurred.

The possible values of this bit are:

**0**  
Error has not occurred.

**1**  
Error has occurred.

This bit resets to zero on a Warm reset.

**INVSTATE, bit [1]**

Invalid state flag. Sticky flag indicating whether an EPSR.T or EPSR.IT validity error has occurred.

The possible values of this bit are:

**0**  
Error has not occurred.

**1**  
Error has occurred.

This bit resets to zero on a Warm reset.

**UNDEFINSTR, bit [0]**

**UNDEFINED** instruction flag. Sticky flag indicating whether an **UNDEFINED** instruction error has occurred.

The possible values of this bit are:

**0**

Error has not occurred.

**1**

Error has occurred.

This includes attempting to execute an **UNDEFINED** instruction associated with an enable coprocessor.

This bit resets to zero on a Warm reset.



### D1.2.217 VTOR, Vector Table Offset Register

The VTOR characteristics are:

#### Purpose

Holds the vector table address for the selected Security state.

#### Usage constraints

Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are **UNPREDICTABLE**.

#### Configurations

This register is always implemented.

#### Attributes

32-bit read/write register located at 0xE000ED08.

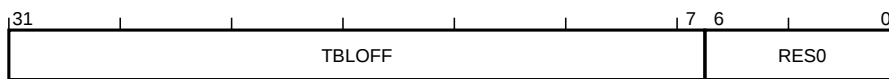
To allow lock down of this register it is **IMPLEMENTATION DEFINED** whether this register is writable.

Secure software can access the Non-secure version of this register via VTOR\_NS located at 0xE002ED08. The location 0xE002ED08 is **RES0** to software executing in Non-secure state and the debugger.

This register is banked between Security states.

### Field descriptions

The VTOR bit assignments are:



#### TBLOFF, bits [31:7]

Table offset. Bits [31:7] of the vector table address for the selected Security state.

It is **IMPLEMENTATION DEFINED** whether any of the TBLOFF bits are WI

This field resets to an **IMPLEMENTATION DEFINED** value on a Warm reset.

#### Bits [6:0]

Reserved, **RES0**.

### D1.2.218 XPSR, Combined Program Status Registers

The XPSR characteristics are:

**Purpose**

Provides access to a combination of the APSR, EPSR and IPSR.

**Usage constraints**

Privileged access only. Unprivileged access is RAZ/WI, unless otherwise stated.

**Configurations**

This register is always implemented.

**Attributes**

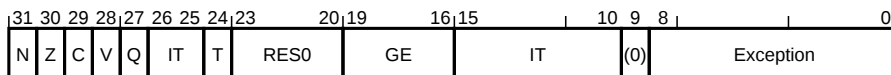
32-bit read/write special-purpose register.

This register is not banked between Security states.

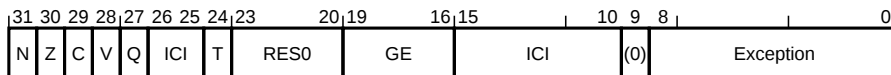
#### Field descriptions

The XPSR bit assignments are:

**When {XPSR[26:25], XPSR[11:10]} != 0:**



**When {XPSR[26:25], XPSR[11:10]} == 0:**



**N, bit [31]**

Negative flag. Reads or writes the current value of APSR.N.

**Z, bit [30]**

Zero flag. Reads or writes the current value of APSR.Z.

**C, bit [29]**

Carry flag. Reads or writes the current value of APSR.C.

**V, bit [28]**

Overflow flag. Reads or writes the current value of APSR.V.

**Q, bit [27]**

Saturate flag. Reads or writes the current value of APSR.Q.

**T, bit [24]**

T32 state. Reads or writes the current value of EPSR.T.

**Bits [23:20]**

Reserved, RES0.

**GE, bits [19:16]**

Greater-than or equal flag. Reads or writes the current value of APSR.GE.

**IT, bits [15:10,26:25] , when [{XPSR[26:25], XPSR[11:10]} != 0]**

If-then flags. Reads or writes the current value of EPSR.IT.

**ICI, bits [26:25,15:10]** , when [ $\{XPSR[26:25], XPSR[11:10]\} == 0$ ]

Interrupt continuation flags. Reads or writes the current value of EPSR.ICI.

**Bit [9]**

Reserved, [RES0](#).

**Exception, bits [8:0]**

Exception number. Reads or writes the current value of IPSR.Exception.

**Part E**  
**Armv8-M Pseudocode**

## Chapter E1

# Arm Pseudocode Definition

This chapter provides a definition of the pseudocode that this manual uses, and defines some *built-in* functions that the pseudocode uses. It contains the following sections:

[E1.1 About the Arm pseudocode on page 1194.](#)

[E1.2 Data types on page 1195.](#)

[E1.3 Operators on page 1201.](#)

[E1.4 Statements and control structures on page 1207.](#)

[E1.5 Built-in functions on page 1212.](#)

[E1.6 Arm pseudocode definition index on page 1215.](#)

[E1.7 Additional functions on page 1218.](#)

### **Note**

This chapter is not a formal language definition for the pseudocode. It is a guide to help understand the use of Arm pseudocode.

## E1.1 About the Arm pseudocode

The Arm pseudocode provides precise descriptions of some areas of the Arm architecture. This includes description of the decoding and operation of all valid instructions.

The following sections describe the Arm pseudocode in detail:

[E1.2 Data types on page 1195.](#)

[E1.3 Operators on page 1201.](#)

[E1.4 Statements and control structures on page 1207.](#)

[E1.5 Built-in functions on page 1212](#) describes some built-in functions that the pseudocode functions use that this manual describes elsewhere.

[E1.6 Arm pseudocode definition index on page 1215](#) contains the indexes to the pseudocode.

### E1.1.1 General limitations of Arm pseudocode

Because of the limitations inherent in all pseudocode, the Arm pseudocode and pseudocode comments describe only one particular implementation of the architecture. There are several instances where a rule relaxes the behavior that a particular piece of pseudocode describes.

The pseudocode statements `EndOfInstruction()`, `SEE`, `UNDEFINED`, `CONSTRAINED_UNPREDICTABLE`, and `UNPREDICTABLE` indicate behavior that differs from that indicated by the pseudocode being executed. If one of the statements is encountered:

- `CONSTRAINED_UNPREDICTABLE`, and `UNPREDICTABLE` mean earlier behavior indicated by the pseudocode is only specified as occurring to the extent required to determine that the statement is executed. No subsequent behavior that the pseudocode indicates occurs.
- `EndOfInstruction()`, `SEE`, and `UNDEFINED` mean that the pseudocode will terminate execution of the current instruction and pseudocode execution continues from the exception catch.

For more information, see [E1.4.5 Special statements on page 1210](#).

## E1.2 Data types

This section describes:

[E1.2.1 General data type rules](#) .

[E1.2.2 Bitstrings](#) .

[E1.2.3 Integers](#) on page 1196.

[E1.2.4 Reals](#) on page 1196.

[E1.2.5 Booleans](#) on page 1197.

[E1.2.6 Enumerations](#) on page 1197.

[E1.2.7 Structures](#) on page 1198.

[E1.2.8 Tuples](#) on page 1199.

[E1.2.9 Arrays](#) on page 1199.

### E1.2.1 General data type rules

Arm architecture pseudocode is a strongly typed language. Every literal and variable is of one of the following types:

- Bitstring.
- Integer.
- Boolean.
- Real.
- Enumeration.
- Tuple.
- Struct.
- Array.

The syntax of a literal determines its type. A variable can be assigned to without an explicit declaration. The variable implicitly has the type of the assigned value. For example, the following assignments implicitly declare the variables `x`, `y` and `z` to have types integer, bitstring of length 1, and Boolean, respectively.

```
1 x = 1;
2 y = '1';
3 z = TRUE;
```

Variables can also have their types declared explicitly by preceding the variable name with the name of the type. The following example declares explicitly that a variable named `count` is an integer.

```
integer count;
```

This is most often done in function definitions for the arguments and the result of the function.

The remaining subsections describe each data type in more detail.

### E1.2.2 Bitstrings

This section describes the bitstring data type.

## Syntax

`bits`(*N*)

The type name of a bitstring of length '*N*'.

`bit`

A synonym of `bits`(1).

## Description

A bitstring is a finite-length string of 0s and 1s. Each length of bitstring is a different type. The minimum permitted length of a bitstring is 0.

Bitstring constants literals are written as a single quotation mark, followed by the string of 0s and 1s, followed by another single quotation mark. For example, the two constants literals of type `bit` are `'0'` and `'1'`. Spaces can be included in bitstrings for clarity.

The bits in a bitstring are numbered from left to right *N*-1 to 0. This numbering is used when accessing the bitstring using bitslices. In conversions to and from integers, bit *N*-1 is the MSByte and bit 0 is the LSByte. This order matches the order in which bitstrings derived from encoding diagrams are printed.

Every bitstring value has a left-to-right order, with the bits being numbered in standard *little-endian* order. That is, the leftmost bit of a bitstring of length *N* is bit (*N*-1) and its right-most bit is bit 0. This order is used as the most-significant-to-least-significant bit order in conversions to and from integers. For bitstring constants and bitstrings that are derived from encoding diagrams, this order matches the way that they are printed.

Bitstrings are the only concrete data type in pseudocode, corresponding directly to the contents values that are manipulated in registers, memory locations, and instructions. All other data types are abstract.

### E1.2.3 Integers

This section describes the data type for integer numbers.

## Syntax

`integer`

The type name for the integer data type.

## Description

Pseudocode integers are unbounded in size and can be either positive or negative. That is, they are mathematical integers rather than what computer languages and architectures commonly call integers. Computer integers are represented in pseudocode as bitstrings of the appropriate length, and the pseudocode provides functions to interpret those bitstrings as integers.

Integer literals are normally written in decimal form, such as 0, 15, -1234. They can also be written in C-style hexadecimal form, such as `0x55` or `0x80000000`. Hexadecimal integer literals are treated as positive unless they have a preceding minus sign. For example, `0x80000000` is the integer  $+2^{31}$ . If  $-2^{31}$  needs to be written in hexadecimal, it must be written as `-0x80000000`.

### E1.2.4 Reals

This section describes the data type for real numbers.



## Syntax

`real`

The type name for the real data type.

## Description

Pseudocode reals are unbounded in size and precision. That is, they are mathematical real numbers, not computer floating-point numbers. Computer floating-point numbers are represented in pseudocode as bitstrings of the appropriate length, and the pseudocode provides functions to interpret those bitstrings as reals.

Real constant literals are written in decimal form with a decimal point. This means `0` is an integer constant literal, but `0.0` is a real constant literal.

### E1.2.5 Booleans

This section describes the boolean data type.

## Syntax

`boolean`

The type name for the boolean data type.

`TRUE, FALSE`

The two values a boolean variable can take.

## Description

A boolean is a logical `TRUE` or `FALSE` value.

### Note

This is not the same type as `bit`, which is a bitstring of length 1. A boolean can only take on one of two values: `TRUE` or `FALSE`.

### E1.2.6 Enumerations

This section describes the enumeration data type.

## Syntax and examples

`enumeration`

Keyword to define a new enumeration type.

```
enumeration Example {Example_One, Example_Two, Example_Three};
```

A definition of a new enumeration that is called `Example`, which can take on the values `Example_One`, `Example_Two`, `Example_Three`.

## Description

An enumeration is a defined set of named values.

An enumeration must contain at least one named value. A named value must not be shared between enumerations.

Enumerations must be defined explicitly, although a variable of an enumeration type can be declared implicitly by assigning one of the named values to it. By convention, each named value starts with the name of the enumeration followed by an underscore. The name of the enumeration is its *type name*, or *type*, and the named values are its possible *values*.

## E1.2.7 Structures

This section describes the structure data type.

### Syntax and examples

`type`

The keyword that is used to declare the structure data type.

```
type ShiftSpec is (bits(2) shift, integer amount):
```

An example definition for a new structure that is called 'ShiftSpec' that contains a bitstring member that is called 'shift' and an integer member called 'amount'. Structure definitions must not be terminated with a semicolon.

```
ShiftSpec abc;
```

A declaration of a variable that is named 'abc' of type 'ShiftSpec'.

```
abc.shift
```

Syntax to refer to the individual members within the structure variable.

### Description

A structure is a compound data type composed of one or more data items. The data items can be of different data types. This can include compound data types. The data items of a structure are called its members and are named.

In the syntax section, the example defines a structure that is called `ShiftSpec` with two members. The first is a bitstring of length 2 named `shift` and the second is an integer that is named `amount`. After declaring a variable of that type that is named `abc`, the members of this structure are referred to as `abc.shift` and `abc.amount`.

Every definition of a structure creates a different type, even if the number and type of their members are identical. For example:

```
type ShiftSpec1 is (bits(2) shift, integer amount)
```

```
type ShiftSpec2 is (bits(2) shift, integer amount)
```

`ShiftSpec1` and `ShiftSpec2` are two different types despite having identical definitions. This means that the value in a variable of type `ShiftSpec1` cannot be assigned to variable of type `ShiftSpec2`.

#### E1.2.7.1 `_Type` and `_Type`

This subsection describes the data structure types for a particular register or payload.

## Example

```
RETPSR_Type
```

The data structure of type RETPSR.

## Description

By convention `_Type` declares a structure data type for a specific register or payload.

See the individual register descriptions for the fields that apply to a particular data structure.

## E1.2.8 Tuples

This section describes the tuple data type.

### Examples

```
(bits(32) shifter_result, bit shifter_carry_out)
```

An example of the tuple syntax.

```
(shift_t, shift_n) = ('00', 0);
```

An example of assigning values to a tuple.

### Description

A tuple is an ordered set of data items, which are separated by commas and enclosed in parentheses. The items can be of different types and a tuple must contain at least one data item.

Tuples are often used as the return type for functions that return multiple results. For example, in the syntax section, the example tuple is the return type of the function `Shift_C()`, which performs a standard A32/T32 shift or rotation. Its return type is a tuple containing two data items, with the first of type `bits(32)`, and the second of type `bit`.

Each tuple is a separate compound data type. The compound data type is represented as a comma-separated list of ordered data types between parentheses. This means that the example tuple at the start of this section is of type `(bits(32), bit)`. The general principle that types can be implied by an assignment extends to implying the type of the elements in the tuple. For example, in the syntax section, the example assignment implicitly declares:

- `shift_t` to be of type `bits(2)`.
- `shift_n` to be of type `integer`.
- `(shift_t, shift_n)` to be a tuple of type `(bits(2), integer)`.

## E1.2.9 Arrays

This section describes the array data type.

### Syntax

```
array
```

The type name for the array data type.

```
array data_type array_name[A..B];
```

```
array [A..B] of data_type array_name
```

Declaration of an array of type ‘data\_type’, which might be compound data type. It is named ‘array\_name’ and is indexed with an integer range from ‘A’ to ‘B’.

## Description

An array is an ordered set of fixed size containing items of a single data type. This can include compound data types. Pseudocode arrays are indexed by either enumerations or integer ranges. An integer range is represented by the lower inclusive end of the range, then `..`, then the upper inclusive end of the range.

For example:

The following example declares an array of 31 bitstrings of length 64, indexed from 0-30.

```
1 array bits(64) _R[0..30];
```

Arrays are always explicitly declared, and there is no notation for a constant literal array. Arrays always contain at least one element data item, because:

- Enumerations always contain at least one symbolic constant named value.
- Integer ranges always contain at least one integer.

An array declared with an enumeration type as the index must be accessed using enumeration values of that enumeration type. An array declared with an integer range type as the index must be accessed using integer values from that inclusive range. Accessing such an array with an integer value outside of the range is a coding error.

Pseudocode can also contain array-like functions such as `R[i]`, `MemU[address, size]`, or `Elem[vector, i, size]`. These functions package up and abstract additional operations that are normally performed on accesses to the underlying arrays, such as register banking, memory protection, endian-dependent byte ordering, exclusive-access housekeeping and Advanced SIMD element processing. See [E1.4.2 Function and procedure calls on page 1207](#).

## E1.3 Operators

This section describes:

[E1.3.1 Relational operators](#) .

[E1.3.2 Boolean operators](#) .

[E1.3.3 Bitstring operators](#) on page 1202.

[E1.3.4 Arithmetic operators](#) on page 1203.

[E1.3.5 The assignment operator](#) on page 1204.

[E1.3.6 Precedence rules](#) on page 1205.

[E1.3.7 Conditional expressions](#) on page 1205.

[E1.3.8 Operator polymorphism](#) on page 1205.

### E1.3.1 Relational operators

The following operations yield results of type `boolean`.

#### Equality and non-equality

If two variables `x` and `y` are of the same type, their values can be tested for equality by using the expression `x == y` and for non-equality by using the expression `x != y`. In both cases, the result is of type `boolean`.

Both `x` and `y` must be of type `bits(N)`, `real`, `enumeration`, `boolean`, or `integer`. Named values from an `enumeration` can only be compared if they are both from the same `enumeration`. An exception is that a bitstring can be tested for equality with an integer to allow a `d == 15` test.

A special form of comparison is defined with a bitstring literal that can contain bit values `'0'`, `'1'`, and `'x'`. Any bit with value `'x'` is ignored in determining the result of the comparison. For example, if `opcode` is a 4-bit bitstring, the expression `opcode == '1x0x'` matches the values `1000`, `1100`, `1001`, and `1101`. This is known as a bitmask.

#### Note

This special form is permitted in the implied equality comparisons in the `when` parts of `case ... of ...` structures.

#### Comparisons

If `x` and `y` are integers or reals, then `x < y`, `x <= y`, `x > y`, and `x = y` are less than, less than or equal, greater than, and greater than or equal comparisons between them, producing boolean results.

#### E1.3.1.1 Set membership with `IN`

`<expression> IN {<set>}` produces `TRUE` if `<expression>` is a member of `<set>`. Otherwise, it is `FALSE`. `<set>` must be a list of expressions that are separated by commas.

### E1.3.2 Boolean operators

If `x` is a boolean expression, then `!x` is its logical inverse.

If `x` and `y` are boolean expressions, then `x && y` is the result of ANDing them together. As in the C language, if `x` is `FALSE`, the result is determined to be `FALSE` without evaluating `y`.

#### Note

This is known as short circuit evaluation.

If  $x$  and  $y$  are `booleans`, then  $x \ || \ y$  is the result of ORing them together. As in the C language, if  $x$  is `TRUE`, the result is determined to be `TRUE` without evaluating  $y$ .

#### Note

If  $x$  and  $y$  are `booleans` or boolean expressions, then the result of  $x \ != \ y$  is the same as the result of exclusive-ORing  $x$  and  $y$  together. The operator `EOR` only accepts bitstring arguments.

### E1.3.3 Bitstring operators

The following operations can be applied only to bitstrings.

#### Logical operations on bitstrings

If  $x$  is a bitstring, `NOT(x)` is the bitstring of the same length that is obtained by logically inverting every bit of  $x$ .

If  $x$  and  $y$  are bitstrings of the same length,  $x \ \text{AND} \ y$ ,  $x \ \text{OR} \ y$ , and  $x \ \text{EOR} \ y$  are the bitstrings of that same length that is obtained by logically ANDing, logically ORing, and exclusive-ORing corresponding bits of  $x$  and  $y$  together.

#### Bitstring concatenation and slicing

If  $x$  and  $y$  are bitstrings of lengths  $N$  and  $M$  respectively, then  $x:y$  is the bitstring of length  $N + M$  constructed by concatenating  $x$  and  $y$  in left-to-right order.

The bitstring slicing operator addresses specific bits in a bitstring. This can be used to create a new bitstring from extracted bits or to set the value of specific bits. Its syntax is  $x[\text{integer\_list}]$ , where  $x$  is the integer or bitstring being sliced, and `[integer_list]` is a comma-separated list of integers that are enclosed in square brackets. The length of the resulting bitstring is equal to the number of integers in `[integer_list]`. In  $x[\text{integer\_list}]$ , each of the integers in `[integer_list]` must be:

- $\geq 0$ .
- $< \text{Len}(x)$  if  $x$  is a bitstring.

The definition of  $x[\text{integer\_list}]$  depends on whether `integer_list` contains more than one integer:

- If `integer_list` contains more than one integer,  $x[i,j,k,\dots,n]$  is defined to be the concatenation:

```
1 x[i]: x[j]: x[k]:...: x[n]
```

- If `integer_list` consists of just one integer  $i$ ,  $x[i]$  is defined to be:
  - If  $x$  is a bitstring, '0' if bit  $i$  of  $x$  is a zero and '1' if bit  $i$  of  $x$  is a one.
  - If  $x$  is an integer, and  $y$  is the unique integer in the range  $0$  to  $2^{(i+1)}-1$  that is congruent to  $x$  modulo  $2^{(i+1)}$ . Then  $x[i]$  is '0' if  $y < 2^i$  and '1' if  $y \geq 2^i$ .

Loosely, this definition treats an integer as equivalent to a sufficiently long two's complement representation of it as a bitstring.

The notation for a range expression is  $i:j$  with  $i \geq j$  is shorthand for the integers in order from  $i$  down to  $j$ , with both end values included. For example, `instr[31:28]` represents `instr[31, 30, 29, 28]`.

$x[\text{integer\_list}]$  is assignable provided  $x$  is an assignable bitstring and no integer appears more than once in `[integer_list]`. In particular,  $x[i]$  is assignable if  $x$  is an assignable bitstring and  $0 \leq i < \text{Len}(x)$ .

Encoding diagrams for registers frequently show named bits or multi-bit fields. For example, the encoding diagram for the APSR shows its bit[31] as `N`. In such cases, the syntax `APSR.N` is used as a more readable synonym for `APSR[31]` as named bits can be referred to with the same syntax as referring to members of a struct. A

comma-separated list of named bits enclosed in square brackets following the register name allows multiple bits to be addressed simultaneously.

For example, `APSR.[N, C, Q]` is synonymous with `APSR [31, 29, 27]`.

### E1.3.4 Arithmetic operators

Most pseudocode arithmetic is performed on integer or real values, with operands obtained by conversions from bitstrings and results that are converted back to bitstrings. As these data types are the unbounded mathematical types, no issues arise about overflow or similar errors.

#### Unary plus and minus

If  $x$  is an integer or real, then  $+x$  is  $x$  unchanged,  $-x$  is  $x$  with its sign reversed. Both are of the same type as  $x$ .

#### Addition and subtraction

If  $x$  and  $y$  are integers or reals,  $x + y$  and  $x - y$  are their sum and difference. Both are of type `integer` if  $x$  and  $y$  are both of type `integer`, and `real` otherwise.

There are two cases where the types of  $x$  and  $y$  can be different. A bitstring and an integer can be added together to allow the operation `PC + 4`. An integer can be subtracted from a bitstring to allow the operation `PC - 2`.

If  $x$  and  $y$  are bitstrings of the same length  $N$ , so that  $N = \text{Len}(x) = \text{Len}(y)$ , then  $x + y$  and  $x - y$  are the least significant  $N$  bits of the results of converting  $x$  and  $y$  to integers and adding or subtracting them. Signed and unsigned conversions produce the same result:

```

1 x + y = (SInt(x) + SInt(y)) [N-1:0]
2 = (UInt(x) + UInt(y)) [N-1:0]
3 x - y = (SInt(x) - SInt(y)) [N-1:0]
4 = (UInt(x) - UInt(y)) [N-1:0]

```

If  $x$  is a bitstring of length  $N$  and  $y$  is an integer,  $x + y$  and  $x - y$  are the bitstrings of length  $N$  defined by  $x + y = x + y[N-1:0]$  and  $x - y = x - y[N-1:0]$ . Similarly, if  $x$  is an integer and  $y$  is a bitstring of length  $M$ ,  $x + y$  and  $x - y$  are the bitstrings of length  $M$  defined by  $x + y = x[M-1:0] + y$  and  $x - y = x[M-1:0] - y$ .

#### Multiplication

If  $x$  and  $y$  are integers or reals, then  $x * y$  is the product of  $x$  and  $y$ . It is of type `integer` if  $x$  and  $y$  are both of type `integer`, and `real` otherwise.

#### Division and modulo

If  $x$  and  $y$  are reals, then  $x/y$  is the result of dividing  $x$  by  $y$ , and is always of type `real`.

If  $x$  and  $y$  are integers, then  $x \text{ DIV } y$  and  $x \text{ MOD } y$  are defined by:

```

1 x DIV y = RoundDown(x/y)
2 x MOD y = x - y * (x DIV y)

```

It is a pseudocode error to use any of  $x/y$ ,  $x \text{ MOD } y$ , or  $x \text{ DIV } y$  in any context where  $y$  can be zero.

#### Scaling

If  $x$  and  $n$  are of type `integer`, then:

- $x \ll n = \text{RoundDown}(x * 2^n)$ .
- $x \gg n = \text{RoundDown}(x * 2^{(-n)})$ .

## Raising to a power

If  $x$  is an integer or a real and  $n$  is an integer, then  $x^n$  is the result of raising  $x$  to the power of  $n$ , and:

- If  $x$  is of type `integer` then  $x^n$  is of type `integer`.
- If  $x$  is of type `real` then  $x^n$  is of type `real`.

### E1.3.5 The assignment operator

The assignment operator is the `=` character, which assigns the value of the right-hand side to the left-hand side. An assignment statement takes the form:

```
<assignable_expression> = <expression>;
```

This following subsection defines valid expression syntax.

#### General expression syntax

An expression is one of the following:

- A literal.
- A variable, optionally preceded by a data type name to declare its type.
- The word `UNKNOWN` preceded by a data type name to declare its type.
- The result of applying a language-defined operator to other expressions.
- The result of applying a function to other expressions.

Variable names normally consist of alphanumeric and underscore characters, starting with an alphabetic or underscore character.

Each register that is defined in an Arm architecture specification defines a correspondingly named pseudocode bitstring variable, and that variable has the stated behavior of the register. For example, if a bit of a register is defined as `RAZ/WI`, then the corresponding bit of its variable reads as ‘0’ and ignore writes.

An expression like `bits(32) UNKNOWN` indicates that the result of the expression is a value of the given type, but the architecture does not specify what value it is and software must not rely on such values. The value produced must not:

- Return information that cannot be accessed at the current or a lower level of privilege using instructions that are not `UNPREDICTABLE` or `CONSTRAINED UNPREDICTABLE` and do not return `UNKNOWN` values,
- Be promoted as providing any useful information to software.

#### Note

`UNKNOWN` values are similar to the definition of `UNPREDICTABLE`, but do not indicate that the entire architectural state becomes unspecified.

Only the following expressions are assignable. This means that these are the only expressions that can be placed on the left-hand side of an assignment:

- Variables.
- The results of applying some operators to other expressions.

The description of each language-defined operator that can generate an assignable expression specifies the circumstances under which it does so. For example, those circumstances might require that one or more of the expressions the operator operates on is an assignable expression.

- The results of applying array-like functions to other expressions. The description of an array-like function specifies the circumstances under which it can generate an assignable expression.



**Note**

If the right-hand side in an assignment is a function returning a tuple, an item in the assignment destination can be written as `-` to indicate that the corresponding item of the assigned tuple value is discarded. For example:

```
(shifted, -) = LSL_C(operand, amount);
```

The expression on the right-hand side itself can be a tuple. For example:

```
(x, y) = (function_1(), function_2());
```

Every expression has a data type.

- For a literal, this data type is determined by the syntax of the literal.
- For a variable, there are the following possible sources for the data type
  - An optional preceding data type name.
  - A data type the variable was given earlier in the pseudocode by recursive application of this rule.
  - A data type the variable is being given by assignment, either by direct assignment to the variable, or by assignment to a list of which the variable is a member.

It is a pseudocode error if none of these data type sources exists for a variable, or if more than one of them exists and they do not agree about the type.

- For a language-defined operator, the definition of the operator determines the data type.
- For a function, the definition of the function determines the data type.

**E1.3.6 Precedence rules**

The precedence rules for expressions are:

1. Literals, variables, and function invocations are evaluated with higher priority than any operators using their results, but see [E1.3.2 Boolean operators on page 1201](#).
2. Operators on integers follow the normal operator precedence rules of *exponentiation before multiply/divide before add/subtract*, with sequences of multiply/divides or add/subtracts evaluated left-to-right.
3. Other expressions must be parenthesized to indicate operator precedence if ambiguity is possible, but do not need to be if all permitted precedence orders under the type rules necessarily lead to the same result. For example, if *i*, *j* and *k* are integer variables, `i > 0 && j > 0 && k > 0` is acceptable, but `i > 0 && j > 0 || k > 0` is not.

**E1.3.7 Conditional expressions**

If *x* and *y* are two values of the same type and *t* is a value of type `boolean`, then `if t then x else y` is an expression of the same type as *x* and *y* that produces *x* if *t* is `TRUE` and *y* if *t* is `FALSE`.

**E1.3.8 Operator polymorphism**

Operators in pseudocode can be polymorphic, with different functionality when applied to different data types. Each resulting form of an operator has a different prototype definition. For example, the operator `+` has forms that act on various combinations of integers, reals and bitstrings.

[Table E1-1](#) summarizes the operand types valid for each unary operator and the result type. [Table E1-2](#) summarizes the operand types valid for each binary operator and the result type.

**Table E1-1, Result and operand types that are permitted for unary operators.**

| Operator | Operand Type | Result Type |
|----------|--------------|-------------|
| -        | integer      | integer     |
|          | real         | real        |
| NOT      | bits (N)     | bits (N)    |
| !        | boolean      | boolean     |

**Table E1-2, Result and operand types that are permitted for binary operators.**

| Operator     | First operand type | Second operand type | Result type |
|--------------|--------------------|---------------------|-------------|
| ==           | bits (N)           | integer             | boolean     |
|              | bits (N)           | bits (N)            |             |
|              | integer            | integer             |             |
|              | real               | real                |             |
|              | enumeration        | enumeration         |             |
| !=           | boolean            | boolean             | boolean     |
|              | bits (N)           | bits (N)            |             |
|              | integer            | integer             |             |
| <, >         | real               | real                | boolean     |
|              | integer            | integer             |             |
| <=, >=       | real               | real                | integer     |
|              | integer            | integer             |             |
| +, -         | real               | real                | real        |
|              | bits (N)           | bits (N)            | bits (N)    |
|              | bits (N)           | integer             | integer     |
| «, »         | integer            | integer             | integer     |
|              | integer            | integer             | integer     |
| *            | real               | real                | real        |
|              | bits (N)           | bits (N)            | bits (N)    |
| /            | real               | real                | real        |
| DIV          | integer            | integer             | integer     |
| MOD          | integer            | integer             | integer     |
|              | bits (N)           | integer             | integer     |
| &&,          | boolean            | boolean             | boolean     |
| AND, OR, EOR | bits (N)           | bits (N)            | bits (N)    |
| ^            | integer            | integer             | integer     |
|              | real               | integer             | real        |

## E1.4 Statements and control structures

This section describes the statements and program structures available in the pseudocode.

### E1.4.1 Statements and Indentation

A simple statement is either an assignment, a function call, or a procedure call. Each statement must be terminated with a semicolon.

Indentation normally indicates the structure in compound statements. The statements that are contained in structures such as `if... then... else...` or procedure and function definitions are indented more deeply than the statement structure itself. The end of a compound statement structure and their end is indicated by returning to the original indentation level or less.

Indentation is normally done by four spaces for each level. Standard indentation uses four spaces for each level of indent.

### E1.4.2 Function and procedure calls

This section describes how functions and procedures are defined and called in the pseudocode.

#### Procedure and function definitions

A procedure definition has the form:

```
1 <procedure name>(<argument prototypes>
2 <statement 1>;
3 <statement 2>;
4 ...
5 <statement n>;
```

where `<argument prototypes>` consists of zero or more argument definitions, which are separated by commas. Each argument definition consists of a type name followed by the name of the argument.

#### Note

This first definition line is not terminated by a semicolon. This distinguishes it from a procedure call.

A function definition is similar, but also declares the return type of the function:

```
1 <return type> <function name>(<argument prototypes>
2 <statement 1>;
3 <statement 2>;
4 ...
5 <statement n>;
```

Array-like functions are similar, but are written with square brackets and have two forms. These two forms exist because reading from and writing to an array element require different functions. They are frequently used in memory operations. An array-like function definition with a return type is equivalent to reading from an array. For example:

```
1 <return type> <function name>[<argument prototypes>]
2 <statement 1>;
3 <statement 2>;
4 ...
5 <statement n>;
```

Its related function definition with no return type is equivalent to writing to an array. For example:

```
1 <function name>[<argument prototypes>] =<value prototype>
2 <statement 1>;
3 <statement 2>;
4 ...
5 <statement n>;
```

The value prototype determines what data type can be written to the array. The two related functions must share the same name, but the value prototype and return type can be different.

## Procedure calls

A procedure call has the form:

```
1 <procedure_name>(<arguments>;
```

## Return statements

A procedure return has the form: return;

A function return has the form:

```
1 return <expression>;
```

where <expression> is of the type declared in the function prototype line.

## E1.4.3 Conditional control structures

This section describes how conditional control structures are used in the pseudocode.

### **if...then...else...**

In addition to being a ternary operator, a multi-line **if...then...else...** structure can act as a control structure and has the form:

```
1 if <boolean_expression> then
2 <statement 1>;
3 <statement 2>;
4 ...
5 <statement n>;
6
7 elseif <boolean_expression> then
8 <statement a>;
9 <statement b>;
10 ...
11 <statement z>;
12 else
13 <statement A>;
14 <statement B>;
15 ...
16 <statement Z>;
```

The block of lines consisting of **elseif** and its indented statements is optional, and multiple **elseif** blocks can be used.

The block of lines consisting of **else** and its indented statements is optional.

Abbreviated one-line forms can be used when the **then** part, and in the **else** part if it is present, contain only simple statements such as:

```
1 if <boolean_expression> then <statement 1>;
2 if <boolean_expression> then <statement 1>; else <statement A>;
3 if <boolean_expression> then <statement 1>; <statement 2>; else <statement A>;
```

### Note

In these forms, <statement 1>, <statement> 2>, and <statement A> must be terminated by semi-colons. This and > the fact that the **else** part is optional distinguish its use as a > control structure from its use as a ternary operator.

### case...of...

A **case...of...** structure has the form:

```
1 case <expression> of
2 when <literal values1>
3 <statement 1>;
4 <statement 2>;
5 ...
6 <statement n>;
7
8 when <literal values2>
9 <statement 1>;
10 <statement 2>;
11 ...
12 <statement n>;
13
14 ...more "when" groups if required...
15
16 otherwise
17 <statement A>;
18 <statement B>;
19 ...
20 <statement Z>;
```

In this structure, <literal values1> and <literal values2> consist of literal values of the same type as <expression>, separated by commas. There can be additional **when** groups in the structure. Abbreviated one line forms of **when** and **otherwise** parts can be used when they contain only simple statements.

If <expression> has a bitstring type, the literal values can also include bitstring literals containing 'x' bits, known as bitmasks. For details, see [Equality and non-equality](#).

## E1.4.4 Loop control structures

This section describes the three loop control structures that are used in the pseudocode.

### repeat...until...

A **repeat...until...** structure has the form:

```
1 repeat
2 <statement 1>;
3 <statement 2>;
4 ...
5 <statement n>;
6 until <boolean_expression>;
```

It executes the statement block at least once, and the loop repeats until <boolean expression> evaluates to **TRUE**. Variables explicitly declared inside the loop body have scope local to that loop and might not be accessed outside the loop body.

### while...do

A **while...do** structure has the form:

```
1 while <boolean_expression> do
2 <statement 1>;
3 <statement 2>;
4 ...
5 statement n>;
```

It begins executing the statement block only if the boolean expression is true. The loop then runs until the expression is false.

### **for . . .**

A **for . . .** structure has the form:

```
1 for <assignable_expression> = <integer_expr1> to <integer_expr2>
2 <statement 1>;
3 <statement 2>;
4 ...
5 <statement n>;
```

The `<assignable_expression>` is initialized to `<integer_expr1>` and compared to `<integer_expr2>`. If `<integer_expr1>` is less than `<integer_expr2>`, the loop body is executed and the `<assignable_expression>` incremented by one. This repeats until `<assignable_expression>` is more than or equal to `<integer_expr2>`.

There is an alternate form:

```
for <assignable_expression> = <integer_expr1> downto <integer_expr2>
```

where `<integer_expr1>` is decremented after the loop body executes and continues until `<assignable_expression>` is less than or equal than `<integer_expr2>`.

### **Try . . . Catch**

A **try . . . catch** structure has the following form:

```
1 try
2 <statement 1>;
3 <statement 2>;
4 ...
5 <statement n>;
6
7 catch <exception>
8 <statement a>;
9 <statement b>;
10 ...
11 <statement z>;
```

The purpose of this structure is to catch exceptions that are generated by the `try` statements.

## **E1.4.5 Special statements**

This section describes statements with particular architecturally defined behaviors.

### **UNDEFINED**

This subsection describes the statement:

```
UNDEFINED;
```

This statement indicates a pseudocode exception that will be caught by the `try . . . catch` block. When caught, this might result in an UNDEFINSTR UsageFault, NOP or NOCP UsageFault.

## UNPREDICTABLE

This subsection describes the statement:

```
UNPREDICTABLE;
```

This statement indicates a special case that replaces the behavior that is defined by the current pseudocode, apart from behavior that is required to determine that the special case applies. The replacement behavior is UNPREDICTABLE.

## CONSTRAINED UNPREDICTABLE

This subsection describes the statement:

```
CONSTRAINED_UNPREDICTABLE;
```

This statement indicates a special case that replaces the behavior that is defined by the current pseudocode, apart from behavior that is required to determine that the special case applies. The replacement behavior is CONSTRAINED UNPREDICTABLE within the limits defined for each particular case, and might vary.

## SEE . . .

This subsection describes the statement:

```
SEE <reference>;
```

This statement indicates a special case that replaces the behavior that is defined by the current instruction pseudocode, apart from behavior that is required to determine that the special case applies.

It usually refers to another instruction, but can also refer to another encoding or note of the same instruction.

## IMPLEMENTATION DEFINED

This subsection describes the statement:

```
IMPDEF {"<text>"};
```

This statement indicates a special case that provides an IMPLEMENTATION DEFINED value or behavior. An optional <text> field can give more information.

## E1.4.6 Comments

The pseudocode supports two styles of comments:

- `//` starts a comment that is terminated by the end of the line.
- `/*` starts a comment that is terminated by `*/`.

`/**/` statements might not be nested, and the first `*/` ends the comment.

### Note

Comment lines do not require a terminating semicolon.

## E1.5 Built-in functions

This section describes:

[E1.5.1 Bitstring manipulation functions](#) .

[E1.5.2 Arithmetic functions](#) on page 1213.

### E1.5.1 Bitstring manipulation functions

The following bitstring manipulation functions are defined:

#### Bitstring length

If  $x$  is a bitstring:

- The bitstring length function `Len(x)` returns the length of  $x$  as an integer.

#### Bitstring concatenation and replication

If  $x$  is a bitstring and  $n$  is an integer with  $n \geq 0$ :

- `Replicate(x, n)` is the bitstring of length  $n \cdot \text{Len}(x)$  consisting of  $n$  copies of  $x$  concatenated together.
- `Zeros(n) = Replicate('0', n)`.
- `Ones(n) = Replicate('1', n)`.

#### Bitstring count

If  $x$  is a bitstring, `BitCount(x)` is an integer result equal to the number of bits of  $x$  that are ones.

#### Testing a bitstring for being all zero or all ones

If  $x$  is a bitstring:

- `IsZero(x)` produces `TRUE` if all of the bits of  $x$  are zeros and `FALSE` if any of them are ones
- `IsZeroBit(x)` produces `'1'` if all of the bits of  $x$  are zeros and `'0'` if any of them are ones.

`IsOnes(x)` and `IsOnit(x)` work in the corresponding ways. This means:

```
1 IsZero(x) = (BitCount(x) == 0)
2 IsOnes(x) = (BitCount(x) == Len(x))
3 IsZeroBit(x) = if IsZero(x) then '1' else '0'
4 IsOnit(x) = if IsOnes(x) then '1' else '0'
```

#### Lowest and highest set bits of a bitstring

If  $x$  is a bitstring, and  $N = \text{Len}(x)$ :

- `LowestSetBit(x)` is the minimum bit number of any of the bits of  $x$  that are ones. If all of its bits are zeros, `LowestSetBit(x) = N`.
- `HighestSetBit(x)` is the maximum bit number of any of the bits of  $x$  that are ones. If all of its bits are zeros, `HighestSetBit(x) = -1`.
- `CountLeadingZeroBits(x)` is the number of zero bits at the left end of  $x$ , in the range 0 to  $N$ . This means:  
`CountLeadingZeroBits(x) = N - 1 - HighestSetBit(x)`.



- `CountLeadingSignBits(x)` is the number of copies of the sign bit of `x` at the left end of `x`, excluding the sign bit itself, and is in the range 0 to  $N-1$ . This means:

`CountLeadingSignBits(x) = CountLeadingZeroBits(x[N-1:1] EOR x[N-2:0]).`

## Zero-extension and sign-extension of bitstrings

If `x` is a bitstring and `i` is an integer, then `ZeroExtend(x, i)` is `x` extended to a length of `i` bits, by adding sufficient zero bits to its left. That is, if `i == Len(x)`, then `ZeroExtend(x, i) = x`, and if `i > Len(x)`, then:

`ZeroExtend(x, i) = Replicate('0', i-Len(x)): x`

If `x` is a bitstring and `i` is an integer, then `SignExtend(x, i)` is `x` extended to a length of `i` bits, by adding sufficient copies of its leftmost bit to its left. That is, if `i == Len(x)`, then `SignExtend(x, i) = x`, and if `i > Len(x)`, then:

`SignExtend(x, i) = Replicate(TopBit(x), i-Len(x)): x`

It is a pseudocode error to use either `ZeroExtend(x, i)` or `SignExtend(x, i)` in a context where it is possible that `i < Len(x)`.

## Converting bitstrings to integers

If `x` is a bitstring, `SInt()` is the integer whose two's complement representation is `x`.

`UInt()` is the integer whose unsigned representation is `x`.

`Int(x, unsigned)` returns either `SInt(x)` or `UInt(x)` depending on the value of its second argument.

## E1.5.2 Arithmetic functions

This section defines built-in arithmetic functions.

### Absolute value

If `x` is either of type real or integer, `Abs(x)` returns the absolute value of `x`. The result is the same type as `x`.

### Rounding and aligning

If `x` is a real:

- `RoundDown(x)` produces the largest integer `n` so that  $n \leq x$ .
- `RoundUp(x)` produces the smallest integer `n` so that  $n \geq x$ .
- `RoundTowardsZero(x)` produces:
  - `RoundDown(x)` if  $x > 0.0$ .
  - `0` if  $x == 0.0$ .
  - `RoundUp(x)` if  $x < 0.0$ .

If `x` and `y` are both of type **integer**, `Align(x, y) = y * (x DIV y)`, and is of type **integer**.

If `x` is of type **bitstring** and `y` is of type **integer**, `Align(x, y) = (Align(UInt(x), y)) [Len(x)-1:0]`, and is a bitstring of the same length as `x`.

It is a pseudocode error to use either form of `Align(x, y)` in any context where `y` can be 0. In practice, `Align(x, y)` is only used with `y` a constant power of two, and the bitstring form used with  $y = 2^n$  has the effect of producing its argument with its `n` low-order bits forced to zero.

## Maximum and minimum

If  $x$  and  $y$  are integers or reals, then  $\text{Max}(x, y)$  and  $\text{Min}(x, y)$  are their maximum and minimum respectively.  $x$  and  $y$  must both be of type integer or of type real. The function returns a value of the same type as its operands.

## E1.6 Arm pseudocode definition index

This section contains the following tables:

[Table E1-3](#) which contains the pseudocode data types.

[Table E1-4](#) which contains the pseudocode operators.

[Table E1-5](#) which contains the pseudocode keywords and control structures.

[Table E1-6](#) which contains the statements with special behaviors.

### Table E1-3 Index of pseudocode data types

| <b>Keyword</b>           | <b>Meaning</b>                                    |
|--------------------------|---------------------------------------------------|
| <code>array</code>       | Type name for the array type                      |
| <code>bit</code>         | Keyword equivalent to <code>bits(1)</code>        |
| <code>bits(N)</code>     | Type name for the bitstring of length N data type |
| <code>boolean</code>     | Type name for the boolean data type               |
| <code>enumeration</code> | Keyword to define a new enumeration type          |
| <code>integer</code>     | Type name for the integer data type               |
| <code>real</code>        | Type name for the real data type                  |
| <code>type</code>        | Keyword to define a new structure                 |

**Table E1-4 Index of pseudocode operators**

| <b>Operator</b> | <b>Meaning</b>                                                                                                                                                     |
|-----------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| -               | Unary minus on integers or reals<br>Subtraction of integers, reals, and bitstrings<br>used in the left-hand side of an assignment or a tuple to discard the result |
| +               | Unary plus on integers or reals<br>Addition of integers, reals, and bitstrings                                                                                     |
| .               | Extract named member from a list<br>Integer in bitstring extraction operator                                                                                       |
| :               | Bitstring concatenation<br>Integer range in bitstring extraction operator                                                                                          |
| !               | Boolean NOT                                                                                                                                                        |
| !=              | Comparison for inequality                                                                                                                                          |
| (...)           | Around arguments of procedure or function                                                                                                                          |
| [...]           | Around array index<br>Around arguments of array-like function                                                                                                      |
| *               | Multiplication of integers, reals and bitstrings                                                                                                                   |
| /               | Division of integers and reals (real result)                                                                                                                       |
| &&              | Boolean AND                                                                                                                                                        |
| <               | <i>Less than</i> comparison of integers and reals                                                                                                                  |
| [...]           | Slicing of specified bits or bitstring or integer                                                                                                                  |
| <<              | Multiply integer by power of 2 (with rounding towards infinity)                                                                                                    |
| <=              | <i>Less than or equal</i> comparison of integers and reals                                                                                                         |
| =               | Assignment operator                                                                                                                                                |
| ==              | Comparison for equality                                                                                                                                            |
| >               | <i>Greater than</i> comparison of integers and reals                                                                                                               |
| >=              | <i>Greater than or equal</i> comparison of integers and reals                                                                                                      |
| >>              | Divide integer by power of 2                                                                                                                                       |
|                 | Boolean OR                                                                                                                                                         |
| ^               | Exponential operator                                                                                                                                               |
| AND             | Bitwise AND of bitstrings                                                                                                                                          |
| DIV             | Quotient from integer division                                                                                                                                     |
| EOR             | Bitwise EOR of bitstrings                                                                                                                                          |
| IN              | Test membership of a certain expression in a set of values                                                                                                         |
| MOD             | Remainder from integer division                                                                                                                                    |
| NOT             | Bitwise inversion of bitstrings                                                                                                                                    |
| OR              | Bitwise OR of bitstrings                                                                                                                                           |

**Table E1-5 Index of pseudocode keywords and control structures**

| <b>Operator</b>                  | <b>Meaning</b>                                                                              |
|----------------------------------|---------------------------------------------------------------------------------------------|
| <code>/*...*/</code>             | Comment delimiters                                                                          |
| <code>//</code>                  | Introduces comment terminated by end of line                                                |
| <code>case...of...</code>        | Control structure                                                                           |
| <code>FALSE</code>               | One of two values a boolean can take (other than <code>TRUE</code> )                        |
| <code>for...=...to...</code>     | Loop control structure, counting up from the initial value to the upper limit               |
| <code>for...=...downto...</code> | Loop control structure, counting down from the initial value to the lower limit             |
| <code>if...then...else...</code> | Condition expression selecting between two values                                           |
| <code>if...then...else...</code> | Conditional control structure                                                               |
| <code>otherwise</code>           | Introduces default in <code>case...of...</code> control structure                           |
| <code>repeat...until...</code>   | Loop control structure that runs at least once until the termination condition is satisfied |
| <code>return</code>              | Procedure or function return                                                                |
| <code>TRUE</code>                | One of two values a boolean can take (other than <code>FALSE</code> )                       |
| <code>try...catch</code>         | Control structure                                                                           |
| <code>when</code>                | Introduces a specific case in <code>case...of...</code> control structure                   |
| <code>while...do...</code>       | Loop control structure that runs until the termination condition is satisfied               |

**Table E1-6 Index of special statements**

| <b>Keyword</b>                         | <b>Meaning</b>                                          |
|----------------------------------------|---------------------------------------------------------|
| <code>IMPLEMENTATION_DEFINED</code>    | Describes <code>IMPLEMENTATION_DEFINED</code> behavior. |
| <code>SEE</code>                       | Points to other pseudocode to use instead               |
| <code>UNDEFINED</code>                 | Cause Undefined Instruction exception                   |
| <code>UNKNOWN</code>                   | Unspecified value                                       |
| <code>CONSTRAINED_UNPREDICTABLE</code> | Unspecified behavior within limits                      |
| <code>UNPREDICTABLE</code>             | Unspecified behavior                                    |

## E1.7 Additional functions

The following functions are not listed in E2 Pseudocode specification, and are only described in this section.

### E1.7.1 `IsSee()`

`IsSee()` returns TRUE if the exception variable that is passed to it was created because all the encodings that matched the instruction that was being decoded called SEE.

See [SEE...](#)

### E1.7.2 `IsUndefined()`

`IsUndefined()` returns TRUE if the exception variable that is passed to it was created because either the instruction that was being decoded did not match any known encoding, or because one of the encodings that was matched called the special statement UNDEFINED.

See [UNDEFINED](#).

## Chapter E2

# Pseudocode Specification

This chapter specifies the Armv8-M pseudocode. It contains the following section:

[Alphabetical Pseudocode List](#)

## E2.1 Alphabetical Pseudocode List

### E2.1.1 `_D`

```
1 // The 32-bit extension register bank for the FP extension.
2
3 array bits(64) _D[0..15];
```

### E2.1.2 `_ITStateChanged`

```
1 // Indicates a write to ITSTATE
2
3 boolean _ITStateChanged;
```

### E2.1.3 `_Mem`

```
1 // _Mem[] - non-assignment (read) form
2 // =====
3 // Perform single-copy atomic, aligned, little-endian read from physical memory
4
5 (boolean, bits(8*size)) _Mem(AddressDescriptor memaddrdesc, integer size)
6 assert size == 1 || size == 2 || size == 4;
7
8 // _Mem[] - assignment (write) form
9 // =====
10 // Perform single-copy atomic, aligned, little-endian write to physical memory
11
12 boolean _Mem(AddressDescriptor memaddrdesc, integer size, bits(8*size) value)
13 assert size == 1 || size == 2 || size == 4;
```

### E2.1.4 `_NextInstrAddr`

```
1 // Address of next instruction to be fetched in case of branch type operation
2
3 bits(32) _NextInstrAddr;
```

### E2.1.5 `_NextInstrITState`

```
1 // Updated ITSTATE for next instruction
2
3 ITSTATEType _NextInstrITState;
```

### E2.1.6 `_PCChanged`

```
1 // Indicates a change in instruction fetch address due to branch type operations
2
3 boolean _PCChanged;
```

### E2.1.7 `_PendingReturnOperation`

```
1 // Indicate any pending exception returns
2
3 boolean _PendingReturnOperation;
```

### E2.1.8 `_R`



```

1 // The physical array of core registers.
2 // _R[RName_PC] is defined to be the address of the current instruction.
3 // The offset of 4 bytes is applied to it by the register access functions.
4
5 array bits(32) _R[RName];

```

### E2.1.9 \_SP

```

1 // _SP()
2 // =====
3
4 // Non-assignment form
5
6 bits(32) _SP(RName spreg)
7 assert ((spreg == RNameSP_Main_NonSecure) ||
8 ((spreg == RNameSP_Main_Secure) && HaveSecurityExt()) ||
9 (spreg == RNameSP_Process_NonSecure) ||
10 ((spreg == RNameSP_Process_Secure) && HaveSecurityExt()));
11
12 return _R[spreg][31:2]:'00';
13
14 // Assignment form
15
16 ExcInfo _SP(RName spreg, boolean excEntry, bits(32) value)
17 excInfo = DefaultExcInfo();
18 (limit, applylimit) = LookUpSPLim(spreg);
19 if applylimit && (UInt(value) < UInt(limit)) then
20 // If the stack limit is violated during exception entry then the stack
21 // pointer is set to the limit value. This both prevents violations and
22 // ensures that the stack pointer is 8 byte aligned.
23 if excEntry then
24 _R[spreg] = limit;
25
26 // Raise the appropriate exception and syndrome information
27 if HaveMainExt() then
28 UFSR.STKOF = '1';
29 // Create the exception. NOTE: If Main Extension is not implemented the fault always
30 // escalates to HardFault.
31 excInfo = CreateException(UsageFault, FALSE, boolean UNKNOWN);
32 if !excEntry then
33 HandleException(excInfo);
34 else
35 // Stack pointer only updated normally if limit not violated
36 _R[spreg] = value[31:2]:'00';
37 return excInfo;

```

### E2.1.10 Abs

```

1 // Abs()
2 // =====
3
4 integer Abs(integer x)
5 return if x >= 0 then x else -x;
6
7 real Abs(real x)
8 return if x >= 0.0 then x else -x;

```

### E2.1.11 AccessAttributes

```

1 // Memory access attributes
2
3 type AccessAttributes is (
4 boolean iswrite, // TRUE for memory stores, FALSE for load accesses
5 boolean ispriv, // TRUE if the access is privileged, FALSE if unprivileged
6 AccType acctype
7)

```

### E2.1.12 AccType

```

1 // Memory reference access type
2 enumeration AccType { AccType_NORMAL, // Normal loads and stores
3 AccType_ORDERED, // Load-Acquire and Store-Release
4 AccType_STACK, // HW generated stacking / unstacking operation
5 AccType_LAZYFP, // HW generated stacking due to lazy
6 // floating point state preservation
7 AccType_IFETCH, // Instruction fetch
8 AccType_VECTABLE // Vector table fetch
9 };

```

### E2.1.13 ActivateException

```

1 // ActivateException()
2 // =====
3
4 ActivateException(integer exceptionNumber, boolean excIsSecure)
5 // If the exception is Secure, directly entry the Secure state.
6 CurrentState = if excIsSecure
7 then SecurityState_Secure else SecurityState_NonSecure;
8 IPSR.Exception = exceptionNumber[8:0]; // Update IPSR to this exception. This
9 // also
10 // causes a transition to privileged
11 // handler
12 // mode as IPSR.Exception != 0
13
14 if HaveMainExt() then
15 ITSTATE = Zeros(8); // IT/ICI bits cleared
16 // PRIMASK, FAULTMASK, BASEPRI unchanged on exception entry
17 if HaveFPEExt() then
18 CONTROL.FPCA = '0'; // Floating-point Extension only
19 CONTROL.S.SFPA = '0';
20 CONTROL.SPSEL = '0'; // CONTROL.SPSEL is updated to indicate
21 // the
22 // selection of the Main stack pointer -
23 // SP_main
24 // CONTROL.nPRIV unchanged
25
26 // Transition exception from pending to active
27 SetPending(exceptionNumber, excIsSecure, FALSE);
28 SetActive(exceptionNumber, excIsSecure, TRUE);

```

### E2.1.14 AddressDescriptor

```

1 // Descriptor used to access the underlying memory array
2
3 type AddressDescriptor is (
4 MemoryAttributes memattrs,
5 bits(32) paddress, // Physical Address
6 AccessAttributes accattrs
7)

```

### E2.1.15 AddWithCarry

```

1 // AddWithCarry()
2 // =====
3
4 (bits(N), bit, bit) AddWithCarry(bits(N) x, bits(N) y, bit carry_in)
5 unsigned_sum = UInt(x) + UInt(y) + UInt(carry_in);
6 signed_sum = SInt(x) + SInt(y) + UInt(carry_in);
7 result = unsigned_sum[N-1:0]; // same value as signed_sum[N-1:0]
8 carry_out = if UInt(result) == unsigned_sum then '0' else '1';
9 overflow = if SInt(result) == signed_sum then '0' else '1';
10 return (result, carry_out, overflow);

```

**E2.1.16 Align**

```

1 // Align()
2 // =====
3
4 integer Align(integer x, integer y)
5 return y * (x DIV y);
6
7 bits(N) Align(bits(N) x, integer y)
8 return Align(UInt(x), y) [N-1:0];

```

**E2.1.17 ALUWritePC**

```

1 // ALUWritePC()
2 // =====
3
4 ALUWritePC(bits(32) address)
5 BranchWritePC(address);

```

**E2.1.18 ASR**

```

1 // ASR()
2 // =====
3
4 bits(N) ASR(bits(N) x, integer shift)
5 assert shift >= 0;
6 if shift == 0 then
7 result = x;
8 else
9 (result, -) = ASR_C(x, shift);
10 return result;

```

**E2.1.19 ASR\_C**

```

1 // ASR_C()
2 // =====
3
4 (bits(N), bit) ASR_C(bits(N) x, integer shift)
5 assert shift > 0;
6 extended_x = SignExtend(x, shift+N);
7 result = extended_x[shift+N-1:shift];
8 carry_out = extended_x[shift-1];
9 return (result, carry_out);

```

**E2.1.20 BigEndian**

```

1 // BigEndian()
2 // =====
3
4 boolean BigEndian()
5 return (AIRCR.ENDIANNESS == '1');

```

**E2.1.21 BigEndianReverse**

```

1 // BigEndianReverse()
2 // =====
3
4 bits(8*N) BigEndianReverse (bits(8*N) value, integer N)
5 assert N == 1 || N == 2 || N == 4;
6 bits(8*N) result;
7 case N of
8 when 1

```

```

9 result[7:0] = value[7:0];
10 when 2
11 result[15:8] = value[7:0];
12 result[7:0] = value[15:8];
13 when 4
14 result[31:24] = value[7:0];
15 result[23:16] = value[15:8];
16 result[15:8] = value[23:16];
17 result[7:0] = value[31:24];
18 return result;

```

### E2.1.22 BitCount

```

1 // BitCount()
2 // =====
3
4 integer BitCount(bits(N) x)
5 integer result = 0;
6 for i = 0 to N-1
7 if x[i] == '1' then
8 result = result + 1;
9 return result;

```

### E2.1.23 BKPTInstrDebugEvent

```

1 // BKPTInstrDebugEvent()
2 // =====
3 // Generates a debug event based on BKPT Instruction.
4
5 BKPTInstrDebugEvent()
6 if !GenerateDebugEventResponse() then
7 excInfo = CreateException(HardFault, FALSE, boolean UNKNOWN);
8 HandleException(excInfo);

```

### E2.1.24 BLXWritePC

```

1 // BLXWritePC()
2 // =====
3
4 BLXWritePC(bits(32) address, boolean allowNonSecure)
5 // If in the Secure state and transitions to the Non-secure state are allowed
6 // then the target state is specified by the LSB of the target address
7 if HaveSecurityExt() && allowNonSecure && IsSecure() then
8 BranchToNS(address);
9 else
10 EPSR.T = address[0];
11 // If EPSR.T == 0 then an exception is taken on the next
12 // instruction: UsageFault('Invalid State') if the Main Extension is
13 // implemented; HardFault otherwise
14
15 BranchTo(address[31:1]:'0');

```

### E2.1.25 BranchTo

```

1 // BranchTo()
2 // =====
3
4 BranchTo(bits(32) address)
5 // Sets the address to fetch the next instruction from. NOTE: The current PC
6 // is not changed directly as this would modify the result of
7 // ThisInstrAddr(), which would cause the wrong return addresses to be used
8 // for some types of exception. The actual update of the PC is done in the
9 // InstructionAdvance() function after the instruction finishes executing.
10 _NextInstrAddr = address;

```

```

11 _PCChanged = TRUE;
12 // Clear any pending exception returns
13 _PendingReturnOperation = FALSE;
14 return;

```

### E2.1.26 BranchToAndCommit

```

1 // BranchToAndCommit()
2 // =====
3
4 BranchToAndCommit(bits(32) address)
5 // This function directly commits the change to the PC, so ThisInstrAddr()
6 // and NextInstrAddr() both point to the target address. Used for exception
7 // returns and resets so the state is consistent before the next instruction
8 // (or exception) is taken.
9 _R[RName_PC] = address[31:1]:'0';
10 _PCChanged = TRUE;
11 _NextInstrAddr = address[31:1]:'0';
12 // Clear any pending exception returns
13 _PendingReturnOperation = FALSE;
14 return;

```

### E2.1.27 BranchToNS

```

1 // BranchToNS()
2 // =====
3 // Branch to an address with an option to change from Secure to Non-secure
4 // state, if currently in Secure state and transition to Non-secure state is allowed.
5 // Transition to Non-secure state is specified by the LSB bit of target
6 // address (address[0]).
7
8 BranchToNS(bits(32) address)
9 assert HaveSecurityExt() && IsSecure();
10 EPSR.T = '1';
11 if address[0] == '0' then
12 CurrentState = SecurityState_NonSecure;
13 if HaveFPEExt() then CONTROL_S.SFPA = '0';
14 BranchTo(address[31:1]:'0');

```

### E2.1.28 BranchWritePC

```

1 // BranchWritePC()
2 // =====
3
4 BranchWritePC(bits(32) address)
5 BranchTo(address[31:1]:'0');

```

### E2.1.29 BXWritePC

```

1 // BXWritePC()
2 // =====
3
4 ExcInfo BXWritePC(bits(32) address, boolean allowNonSecure)
5 exc = DefaultExcInfo();
6 if HaveSecurityExt() && address == '1111 1110 1111 1111 1111 1111 1111 111x' then
7 // Unlike exception return, any faults raised during a FNC_RETURN
8 // unstacking are raised synchronously with the instruction that triggered
9 // the unstacking.
10 exc = FunctionReturn();
11
12 elseif CurrentMode() == PEMode_Handler && address[31:24] == '11111111' then
13 // The actual exception return is performed when the
14 // current instruction completes. This is because faults that occur
15 // during the exception return are handled differently from faults

```

```

16 // raised during the instruction execution.
17 PendReturnOperation(address);
18
19 elsif HaveSecurityExt() && IsSecure() && allowNonSecure then
20 // If in the Secure state and transitions to the Non-secure state are allowed
21 // then the target state is specified by the LSB of the target address
22 BranchToNS(address);
23
24 else
25 EPSR.T = address[0];
26 // If EPSR.T == 0 then an exception is taken on the next
27 // instruction: UsageFault('Invalid State') if the Main Extension is
28 // implemented; HardFault otherwise
29 BranchTo(address[31:1]:'0');
30
31 return exc;

```

### E2.1.30 CallSupervisor

```

1 // CallSupervisor()
2 // =====
3
4 CallSupervisor()
5 excInfo = CreateException(SVCall, FALSE, boolean UNKNOWN);
6 HandleException(excInfo);

```

### E2.1.31 CanHaltOnEvent

```

1 // CanHaltOnEvent()
2 // =====
3
4 boolean CanHaltOnEvent(boolean is_secure)
5 if !HaveSecurityExt() then assert !is_secure;
6 return (HaveHaltingDebug() && HaltingDebugAllowed() && DHCSR.C_DEBUGEN == '1' &&
7 DHCSR.S_HALT == '0' && (!is_secure || DHCSR.S_SDE == '1'));

```

### E2.1.32 CanPendMonitorOnEvent

```

1 // CanPendMonitorOnEvent()
2 // =====
3
4 boolean CanPendMonitorOnEvent(boolean is_secure, boolean check_pri)
5 if !HaveSecurityExt() then assert !is_secure;
6 return (HaveDebugMonitor() && !CanHaltOnEvent(is_secure) && DEMCR.MON_EN == '1' &&
7 DHCSR.S_HALT == '0' && (!is_secure || DEMCR.SDME == '1') &&
8 (!check_pri || ExceptionPriority(DebugMonitor, is_secure, TRUE) <
9 ExecutionPriority()));

```

### E2.1.33 CheckCPEnabled

```

1 // CheckCPEnabled()
2 // =====
3
4 ExcInfo CheckCPEnabled(integer cp, boolean privileged, boolean secure)
5 (enabled, toSecure) = IsCPEnabled(cp, privileged, secure);
6 if !enabled then
7 if toSecure then
8 UFSR_S.NOCP = '1';
9 else
10 UFSR_NS.NOCP = '1';
11 excInfo = CreateException(UsageFault, TRUE, toSecure);
12 else
13 excInfo = DefaultExcInfo();
14 return excInfo;

```

```

15
16 ExcInfo CheckCPEnabled(integer cp)
17 return CheckCPEnabled(cp, CurrentModeIsPrivileged(), IsSecure());

```

### E2.1.34 CheckDecodeFaults

```

1 // CheckDecodeFaults()
2 // =====
3 // Check and raise faults in the correct order
4
5 CheckDecodeFaults(boolean dp_operation)
6 // Check FP Extension is supported, else raise NOCP Fault
7 if !HaveFPEExt() then
8 secure = IsSecure();
9 if secure then
10 UFSR_S.NOCP = '1';
11 else
12 UFSR_NS.NOCP = '1';
13 excInfo = CreateException(UsageFault, TRUE, secure);
14 HandleException(excInfo);
15
16 // Check access to FP coprocessor is enabled, else raise NOCP Fault
17 excInfo = CheckCPEnabled(10);
18 HandleException(excInfo);
19
20 if dp_operation && HaveSPFPOnly() then UNDEFINED;
21
22
23 CheckDecodeFaults()
24 CheckDecodeFaults(FALSE);

```

### E2.1.35 CheckPermission

```

1 // CheckPermission()
2 // =====
3
4 ExcInfo CheckPermission(Permissions perms, bits(32) address, AccType acctype,
5 boolean iswrite, boolean ispriv, boolean isSecure)
6 if !perms.apValid then
7 fault = TRUE;
8 elsif (perms.xn == '1') && (acctype == AccType_IFETCH) then
9 fault = TRUE;
10 else
11 case perms.ap of
12 when '00' fault = !ispriv;
13 when '01' fault = FALSE;
14 when '10' fault = !ispriv || iswrite;
15 when '11' fault = iswrite;
16 otherwise UNPREDICTABLE;
17
18 // If a fault occurred generate the syndrome info and create the exception.
19 if fault then
20 // Create and write out the syndrome info on implementations with Main Extension.
21 if HaveMainExt() then
22 MMFSR_Type fsr = Zeros(8);
23 case acctype of
24 when AccType_IFETCH
25 fsr.IACCVIOL = '1';
26 when AccType_STACK
27 if iswrite then
28 fsr.MSTKERR = '1';
29 else
30 fsr.MUNSTKERR = '1';
31 when AccType_LAZYFP
32 fsr.MLSPERR = '1';
33 when AccType_NORMAL, AccType_ORDERED
34 fsr.MMARVALID = '1';

```

```

35 fsr.DACCVIOL = '1';
36 otherwise
37 assert (FALSE);
38
39 // Write the syndrome information to the correct instance of banked
40 // registers
41 if isSecure then
42 MMFSR_S = MMFSR_S OR fsr;
43 if fsr.MMARVALID == '1' then
44 MMFAR_S = address;
45 else
46 MMFSR_NS = MMFSR_NS OR fsr;
47 if fsr.MMARVALID == '1' then
48 MMFAR_NS = address;
49
50 // Create the exception. NOTE: If Main Extension is not implemented the fault
51 // escalates to a HardFault
52 excInfo = CreateException(MemManage, TRUE, isSecure);
53 else
54 excInfo = DefaultExcInfo();
55 return excInfo;

```

### E2.1.36 ClearEventRegister

```

1 // ClearEventRegister
2 // =====
3 // Clears the Event register
4
5 ClearEventRegister();

```

### E2.1.37 ClearExclusiveByAddress

```

1 // ClearExclusiveByAddress
2 // =====
3 // Clear the global exclusive monitor for all PEs, except for the PE specified
4 // by processorid for which an address region including any of size bytes
5 // starting from address has had a request for an exclusive access
6
7 ClearExclusiveByAddress(bits(32) address, integer exclprocessorid, integer size);

```

### E2.1.38 ClearExclusiveLocal

```

1 // ClearExclusiveLocal()
2 // =====
3 // Clear local exclusive monitor records for the PE.
4
5 ClearExclusiveLocal(integer processorid);

```

### E2.1.39 ComparePriorities

```

1 // ComparePriorities()
2 // =====
3
4 boolean ComparePriorities(integer exc0Pri, integer exc0Number, boolean exc0IsSecure,
5 integer exc1Pri, integer exc1Number, boolean exc1IsSecure)
6 if exc0Pri != exc1Pri then
7 takeE0 = exc0Pri < exc1Pri;
8 elseif exc0Number != exc1Number then
9 takeE0 = exc0Number < exc1Number;
10 elseif exc0IsSecure != exc1IsSecure then
11 takeE0 = exc0IsSecure;
12 else
13 // The two exceptions have exactly the same priority, so exception 0
14 // cannot be taken in preference to exception 1.

```



```

15 takeE0 = FALSE;
16 return takeE0;
17
18
19 boolean ComparePriorities(ExcInfo exc0Info, boolean groupPri,
20 integer exc1Pri, integer exc1Number, boolean exc1IsSecure)
21 exc0Pri = ExceptionPriority(exc0Info.fault, exc0Info.isSecure, groupPri);
22 return ComparePriorities(exc0Pri, exc0Info.fault, exc0Info.isSecure,
23 exc1Pri, exc1Number, exc1IsSecure);

```

### E2.1.40 ConditionHolds

```

1 // ConditionHolds()
2 // =====
3
4 boolean ConditionHolds(bits(4) cond)
5
6 // Evaluate base condition.
7 case cond[3:1] of
8 when '000' result = (APSR.Z == '1'); // EQ or NE
9 when '001' result = (APSR.C == '1'); // CS or CC
10 when '010' result = (APSR.N == '1'); // MI or PL
11 when '011' result = (APSR.V == '1'); // VS or VC
12 when '100' result = (APSR.C == '1') && (APSR.Z == '0'); // HI or LS
13 when '101' result = (APSR.N == APSR.V); // GE or LT
14 when '110' result = (APSR.N == APSR.V) && (APSR.Z == '0'); // GT or LE
15 when '111' result = TRUE; // AL
16
17 // Condition flag values in the set '111x' indicate the instruction is always executed.
18 // Otherwise, invert condition if necessary.
19 if cond[0] == '1' && cond != '1111' then
20 result = !result;
21
22 return result;

```

### E2.1.41 ConditionPassed

```

1 // ConditionPassed()
2 // =====
3
4 boolean ConditionPassed()
5 return ConditionHolds(CurrentCond());

```

### E2.1.42 ConstrainUnpredictableBool

```

1 // ConstrainUnpredictableBool()
2 // =====
3 // This is a wrapper for UNPREDICTABLE cases where the constrained result is
4 // either TRUE or FALSE.
5
6 boolean ConstrainUnpredictableBool(Unpredictable which);

```

### E2.1.43 ConsumeExcStackFrame

```

1 // ConsumeExcStackFrame()
2 // =====
3
4 ConsumeExcStackFrame(EXC_RETURN_Type excReturn, bit fourByteAlign)
5 // Calculate the size of the integer part of the stack frame
6 toSecure = HaveSecurityExt() && excReturn[6] == '1';
7 if toSecure && (excReturn.ES == '0' ||
8 excReturn.DCRS == '0') then
9 framesize = 0x48;
10 else

```

```

11 framesize = 0x20;
12 // Add on the size of the FP part of the stack frame if present
13 if HaveFPExt() && excReturn.FType == '0' then
14 if toSecure && FPCCR_S.TS == '1' then
15 framesize = framesize + 0x88;
16 else
17 framesize = framesize + 0x48;
18
19 // Update stack pointer. NOTE: Stack pointer limit not checked on exception
20 // return as stack pointer guaranteed to be ascending not descending.
21 mode = if excReturn.Mode == '1' then PMode_Thread else PMode_Handler;
22 spName = LookUpSP_with_security_mode(toSecure, mode);
23 _R[spName] = (_SP(spName) + framesize) OR ZeroExtend(fourByteAlign:'00',32);

```

#### E2.1.44 ConsumptionOfSpeculativeDataBarrier

```

1 // Consumption of Speculative Data Barrier
2 // =====
3 // Perform a Consumption of Speculative Data Barrier operation.
4
5 ConsumptionOfSpeculativeDataBarrier();

```

#### E2.1.45 Coproc\_Accepted

```

1 // Coproc_Accepted
2 // =====
3 // Check whether a coprocessor accepts an instruction.
4
5 boolean Coproc_Accepted(integer cp_num, bits(32) instr);

```

#### E2.1.46 Coproc\_DoneLoading

```

1 // Coproc_DoneLoading
2 // =====
3 // Check whether enough 32-bit words have been loaded for an LDC instruction
4
5 boolean Coproc_DoneLoading(integer cp_num, bits(32) instr);

```

#### E2.1.47 Coproc\_DoneStoring

```

1 // Coproc_DoneStoring
2 // =====
3 // Check whether enough 32-bit words have been stored for a STC instruction
4
5 boolean Coproc_DoneStoring(integer cp_num, bits(32) instr);

```

#### E2.1.48 Coproc\_GetOneWord

```

1 // Coproc_GetOneWord
2 // =====
3 // Gets the 32-bit word for an MRC instruction from the coprocessor
4
5 bits(32) Coproc_GetOneWord(integer cp_num, bits(32) instr);

```

#### E2.1.49 Coproc\_GetTwoWords

```

1 // Coproc_GetTwoWords
2 // =====
3 // Get two 32-bit words for an MRRC instruction from the coprocessor
4
5 (bits(32), bits(32)) Coproc_GetTwoWords(integer cp_num, bits(32) instr);

```

**E2.1.50 Coproc\_GetWordToStore**

```

1 // Coproc_GetWordToStore
2 // =====
3 // Gets the next 32-bit word to store for an STC instruction from the coprocessor
4
5 bits(32) Coproc_GetWordToStore(integer cp_num, bits(32) instr);

```

**E2.1.51 Coproc\_InternalOperation**

```

1 // Coproc_InternalOperation
2 // =====
3 // Instructs a coprocessor to perform the internal operation requested
4 // by a CDP instruction
5
6 Coproc_InternalOperation(integer cp_num, bits(32) instr);

```

**E2.1.52 Coproc\_SendLoadedWord**

```

1 // Coproc_SendLoadedWord
2 // =====
3 // Sends a loaded 32-bit word for an LDC instruction to the coprocessor
4
5 Coproc_SendLoadedWord(bits(32) word, integer cp_num, bits(32) instr);

```

**E2.1.53 Coproc\_SendOneWord**

```

1 // Coproc_SendOneWord
2 // =====
3 // Sends the 32-bit word for an MCR instruction to the coprocessor
4
5 Coproc_SendOneWord(bits(32) word, integer cp_num, bits(32) instr);

```

**E2.1.54 Coproc\_SendTwoWords**

```

1 // Coproc_SendTwoWords
2 // =====
3 // Send two 32-bit words for an MCRR instruction to the coprocessor.
4
5 Coproc_SendTwoWords(bits(32) word2, bits(32) word1, integer cp_num, bits(32) instr);

```

**E2.1.55 CountLeadingSignBits**

```

1 // CountLeadingSignBits()
2 // =====
3
4 integer CountLeadingSignBits(bits(N) x)
5 return CountLeadingZeroBits(x[N-1:1] EOR x[N-2:0]);

```

**E2.1.56 CountLeadingZeroBits**

```

1 // CountLeadingZeroBits()
2 // =====
3
4 integer CountLeadingZeroBits(bits(N) x)
5 return N - 1 - HighestSetBit(x);

```

**E2.1.57 CreateException**

```

1 // CreateException()
2 // =====
3
4 ExcInfo CreateException(integer exception, boolean forceSecurity,
5 boolean isSecure, boolean isSynchronous)
6
7 // Work out the effective target state of the exception
8 if HaveSecurityExt() then
9 if !forceSecurity then
10 isSecure = ExceptionTargetsSecure(exception, IsSecure());
11 else
12 isSecure = FALSE;
13
14 // An implementation without Security Extensions cannot cause a fault targeting
15 // Secure state
16 assert HaveSecurityExt() || !isSecure;
17
18 // Get the remaining exception details
19 (escalateToHf, termInst) = ExceptionDetails(exception, isSecure, isSynchronous);
20
21 // Fill in the default exception info
22 info = DefaultExcInfo();
23 info.fault = exception;
24 info.termInst = termInst;
25 info.origFault = exception;
26 info.origFaultIsSecure = isSecure;
27
28 // Check for HardFault escalation
29 // NOTE: In some cases (for example faults during lazy floating-point state preservation)
30 // the decision to escalate below is ignored and instead based on the info.
31 // origFault*
32 // fields and other factors.
33 if escalateToHf && info.fault != HardFault then
34 // Update the exception info with the escalation details, including
35 // whether there's a change in destination Security state.
36 info.fault = HardFault;
37 isSecure = ExceptionTargetsSecure(HardFault, isSecure);
38 (escalateToHf, -) = ExceptionDetails(HardFault, isSecure, isSynchronous);
39
40 // If the requested exception was already a HardFault then we can't escalate
41 // to a HardFault, so lockup. NOTE: Asynchronous BusFaults never cause
42 // lockups, if the BusFault is disabled it escalates to a HardFault that is
43 // pending.
44 if escalateToHf && isSynchronous && info.fault == HardFault then
45 info.lockup = TRUE;
46
47 // Fill in the remaining exception info
48 info.isSecure = isSecure;
49 return info;
50 ExcInfo CreateException(integer exception, boolean forceSecurity, boolean isSecure)
51 return CreateException(exception, forceSecurity, isSecure, TRUE);

```

### E2.1.58 CurrentCond

```

1 // CurrentCond()
2 // =====
3 // Returns condition specifier of current instruction.
4
5 bits(4) CurrentCond();

```

### E2.1.59 CurrentMode

```

1 // CurrentMode()
2 // =====
3
4 PMode CurrentMode()

```

```
5 return if IPSR.Exception == NoFault[8:0] then PMode_Thread else PMode_Handler;
```

### E2.1.60 CurrentModelsPrivileged

```
1 // CurrentModeIsPrivileged()
2 // =====
3
4 boolean CurrentModeIsPrivileged()
5 return CurrentModeIsPrivileged(IsSecure());
6
7 boolean CurrentModeIsPrivileged(boolean isSecure)
8 nPriv = if isSecure then CONTROL_S.nPRIV else CONTROL_NS.nPRIV;
9 return (CurrentMode() == PMode_Handler || nPriv == '0');
```

### E2.1.61 D

```
1 // D[]
2 // ===
3
4 // Non-assignment form
5
6 bits(64) D[integer n]
7 assert n >= 0 && n <= 31;
8 if n >= 16 && VFPSmallRegisterBank() then UNDEFINED;
9 return _D[n];
10
11 // Assignment form
12
13 D[integer n] = bits(64) value
14 assert n >= 0 && n <= 31;
15 if n >= 16 && VFPSmallRegisterBank() then UNDEFINED;
16 _D[n] = value;
17 return;
```

### E2.1.62 DataMemoryBarrier

```
1 // DataMemoryBarrier()
2 // =====
3 // Perform a Data Memory Barrier operation
4
5 DataMemoryBarrier(bits(4) option);
```

### E2.1.63 DataSynchronizationBarrier

```
1 // DataSynchronizationBarrier
2 // =====
3 // Perform a data synchronization barrier operation
4
5 DataSynchronizationBarrier(bits(4) option);
```

### E2.1.64 DeActivate

```
1 // DeActivate()
2 // =====
3
4 DeActivate(integer returningExceptionNumber, boolean targetDomainSecure)
5 // To prevent the execution priority remaining negative (and therefore
6 // masking HardFault) when returning from NMI / HardFault with a corrupted
7 // IPSR value, the active bits corresponding to the execution priority are
8 // cleared if the raw execution priority (ie the priority before FAULTMASK
9 // and other priority boosting is considered) is negative.
10 rawPri = RawExecutionPriority();
11 if rawPri == -1 then
```

```

12 SetActive(HardFault, AIRCR.BFHFNMINS == '0', FALSE);
13 elseif rawPri == -2 then
14 SetActive(NMI, AIRCR.BFHFNMINS == '0', FALSE);
15 elseif rawPri == -3 then
16 SetActive(HardFault, TRUE, FALSE);
17 else
18 secure = HaveSecurityExt() && targetDomainSecure;
19 SetActive(returningExceptionNumber, secure, FALSE);
20
21 /* PRIMASK and BASEPRI unchanged on exception exit */
22 if HaveMainExt() && rawPri >= 0 then
23 // clear FAULTMASK for exception security domain on any return except
24 // NMI and HardFault
25 if HaveSecurityExt() && targetDomainSecure then
26 FAULTMASK_S[0] = '0';
27 else
28 FAULTMASK_NS[0] = '0';
29 return;

```

### E2.1.65 Debug\_authentication

```

1 // In the recommended CoreSight interface, there are four signals for external debug
2 // authentication, DBGGEN, SPIDEN, NIDEN and SPNIDEN. Each signal is active-HIGH.
3
4 signal DBGGEN;
5 signal SPIDEN;
6 signal NIDEN;
7 signal SPNIDEN;

```

### E2.1.66 DecodeExecute

```

1 // DecodeExecute
2 // =====
3 // Decode instruction and execute
4
5 DecodeExecute(bits(32) instr, bits(32) pc, boolean isT16);

```

### E2.1.67 DecodeImmShift

```

1 // DecodeImmShift()
2 // =====
3
4 (SRTYPE, integer) DecodeImmShift(bits(2) sr_type, bits(5) imm5)
5
6 case sr_type of
7 when '00'
8 shift_t = SRTYPE_LSL; shift_n = UInt(imm5);
9 when '01'
10 shift_t = SRTYPE_LSR; shift_n = if imm5 == '00000' then 32 else UInt(imm5);
11 when '10'
12 shift_t = SRTYPE_ASR; shift_n = if imm5 == '00000' then 32 else UInt(imm5);
13 when '11'
14 if imm5 == '00000' then
15 shift_t = SRTYPE_RRX; shift_n = 1;
16 else
17 shift_t = SRTYPE_ROR; shift_n = UInt(imm5);
18
19 return (shift_t, shift_n);

```

### E2.1.68 DecodeRegShift

```

1 // DecodeRegShift()
2 // =====
3

```

```

4 SRType DecodeRegShift (bits (2) sr_type)
5 case sr_type of
6 when '00' shift_t = SRType_LSL;
7 when '01' shift_t = SRType_LSR;
8 when '10' shift_t = SRType_ASR;
9 when '11' shift_t = SRType_ROR;
10 return shift_t;

```

### E2.1.69 DefaultExcInfo

```

1 // DefaultExcInfo()
2 // =====
3
4 ExcInfo DefaultExcInfo()
5 ExcInfo exc;
6
7 exc.fault = NoFault;
8 exc.origFault = NoFault;
9 exc.isSecure = TRUE;
10 exc.isTerminal = FALSE;
11 exc.inExcTaken = FALSE;
12 exc.lockup = FALSE;
13 exc.termInst = TRUE;
14 return exc;

```

### E2.1.70 DefaultMemoryAttributes

```

1 // DefaultMemoryAttributes()
2 // =====
3
4 MemoryAttributes DefaultMemoryAttributes (bits (32) address)
5
6 MemoryAttributes memattrs;
7
8 case address[31:29] of
9 when '000'
10 memattrs.memtype = MemType_Normal;
11 memattrs.device = DeviceType UNKNOWN;
12 memattrs.innerattrs = '10';
13 memattrs.shareable = FALSE;
14 when '001'
15 memattrs.memtype = MemType_Normal;
16 memattrs.device = DeviceType UNKNOWN;
17 memattrs.innerattrs = '01';
18 memattrs.shareable = FALSE;
19 when '010'
20 memattrs.memtype = MemType_Device;
21 memattrs.device = DeviceType_nGnRE;
22 memattrs.innerattrs = '00';
23 memattrs.shareable = TRUE;
24 when '011'
25 memattrs.memtype = MemType_Normal;
26 memattrs.device = DeviceType UNKNOWN;
27 memattrs.innerattrs = '01';
28 memattrs.shareable = FALSE;
29 when '100'
30 memattrs.memtype = MemType_Normal;
31 memattrs.device = DeviceType UNKNOWN;
32 memattrs.innerattrs = '10';
33 memattrs.shareable = FALSE;
34 when '101'
35 memattrs.memtype = MemType_Device;
36 memattrs.device = DeviceType_nGnRE;
37 memattrs.innerattrs = '00';
38 memattrs.shareable = TRUE;
39 when '110'
40 memattrs.memtype = MemType_Device;

```

```

41 memattrs.device = DeviceType_nGnRE;
42 memattrs.innerattrs = '00';
43 memattrs.shareable = TRUE;
44 when '111'
45 if address[28:20] == '00000000' then
46 memattrs.memtype = MemType_Device;
47 memattrs.device = DeviceType_nGnRnE;
48 memattrs.innerattrs = '00';
49 memattrs.shareable = TRUE;
50 else
51 memattrs.memtype = MemType_Device;
52 memattrs.device = DeviceType_nGnRE;
53 memattrs.innerattrs = '00';
54 memattrs.shareable = TRUE;
55
56 // Outer attributes are the same as the inner attributes in all cases.
57 memattrs.outerattrs = memattrs.innerattrs;
58 memattrs.outershareable = memattrs.shareable;
59
60 // Setting as UNKNOWN by default. This flag will be overwritten based on
61 // SAU/IDAU checking in SecurityCheck()
62 memattrs.NS = boolean UNKNOWN;
63 return memattrs;

```

### E2.1.71 DefaultPermissions

```

1 // DefaultPermissions()
2 // =====
3
4 Permissions DefaultPermissions(bits(32) address)
5
6 Permissions perms;
7
8 perms.ap = '01';
9 perms.apValid = TRUE;
10 perms.region = Zeros(8);
11 perms.regionValid = FALSE;
12
13 case address[31:29] of
14 when '000'
15 perms.xn = '0';
16 when '001'
17 perms.xn = '0';
18 when '010'
19 perms.xn = '1';
20 when '011'
21 perms.xn = '0';
22 when '100'
23 perms.xn = '0';
24 when '101'
25 perms.xn = '1';
26 when '110'
27 perms.xn = '1';
28 when '111'
29 perms.xn = '1';
30
31 return perms;

```

### E2.1.72 DerivedLateArrival

```

1 // DerivedLateArrival()
2 // =====
3
4 DerivedLateArrival(integer pePriority, integer peNumber, boolean peIsSecure, ExcInfo deInfo,
5 integer oeNumber, boolean oeIsSecure)
6 // PE: the pre-empted exception - before exception entry
7 // OE: the original exception - exception entry

```



```

8 // DE: the derived exception - fault on exception entry
9
10 // Get the priorities of the exceptions
11 // xePriority: the lower the value, the higher the priority
12 oePriority = ExceptionPriority(oeNumber, oeIsSecure, FALSE);
13 // NOTE: Comparison of dePriority against PE priority and possible
14 // escalation to HardFault has already occurred. See CreateException().
15
16 // Is the derived exception a DebugMonitor
17 if HaveMainExt() then
18 deIsDbgMonFault = (deInfo.origFault == DebugMonitor);
19 else
20 deIsDbgMonFault = FALSE;
21
22 // Work out which fault to take, and what the target domain is
23 if deInfo.isTerminal then
24 // Derived exception is terminal and prevents the original exception
25 // being taken (eg fault on vector fetch). As a result the derived
26 // exception is treated as a HardFault.
27 targetIsSecure = deInfo.isSecure;
28 targetFault = deInfo.fault;
29 // If the derived fault does not have sufficient priority to pre-empt
30 // lockup instead of taking it.
31 if !ComparePriorities(deInfo, FALSE, oePriority, oeNumber, oeIsSecure) then
32 ActivateException(oeNumber, oeIsSecure);
33 // Since execution of original exception cannot be started, lockup
34 // at the current priority level. That is the priority of the original
35 // exception.
36 Lockup(TRUE);
37 elseif deIsDbgMonFault && !ComparePriorities(deInfo, TRUE, pePriority, peNumber,
38 peIsSecure) then
39 // Ignore the DebugMonitorFault and take original exception
40 SetPending(DebugMonitor, deInfo.isSecure, FALSE);
41 targetFault = oeNumber;
42 targetIsSecure = oeIsSecure;
43 elseif ComparePriorities(deInfo, FALSE, oePriority, oeNumber, oeIsSecure) then
44 // Derive exception has a higher priority (that is a lower value) than the
45 // original exception, so the derived exception first. Tail-chaining
46 // IMPLEMENTATION DEFINED
47 targetFault = deInfo.fault;
48 targetIsSecure = deInfo.isSecure;
49 else
50 // If the derived exception caused a lockup then this must be handled
51 // now as the lockup cannot be pended until the original exception
52 // returns
53 if deInfo.lockup then
54 // Lockup at the priority of the original exception being entered.
55 ActivateException(oeNumber, oeIsSecure);
56 Lockup(TRUE);
57 else
58 // DE will be pended below, start execution of the OE
59 targetFault = oeNumber;
60 targetIsSecure = oeIsSecure;
61
62 // If not of the tests above have triggered a lockup (which would have
63 // terminated execution of the pseudocode) then the derived exception
64 // must be pended and any escalation syndrome info generated
65 if HaveMainExt() &&
66 (deInfo.fault == HardFault) &&
67 (deInfo.origFault != HardFault) then
68 HFSR.FORCED = '1';
69 SetPending(deInfo.fault, deInfo.isSecure, TRUE);
70
71 // Take the target exception. NOTE: None terminal faults are ignored when
72 // handling the derived exception, allowing forward progress to be made.
73 excInfo = ExceptionTaken(targetFault, deInfo.inExcTaken, targetIsSecure, TRUE);
74 // If trying to take the resulting exception results in another fault, then handle
75 // the derived derived fault.
76 if excInfo.fault != NoFault then

```

```

76 DerivedLateArrival(pePriority, peNumber, peIsSecure, excInfo, targetFault,
 targetIsSecure);

```

### E2.1.73 DeviceType

```

1 // Types of memory
2
3 enumeration DeviceType {DeviceType_GRE, DeviceType_nGRE, DeviceType_nGnRE, DeviceType_nGnRnE
 };

```

### E2.1.74 DWT\_AddressCompare

```

1 // DWT_AddressCompare()
2 // =====
3 // Returns a pair of values. The first result is whether the (masked) addresses are equal,
4 // where the access address (addr) is masked according to DWT_FUNCTION[n].DATAVSIZ and the
5 // comparator address (compaddr) is masked according to the access size. The second result
6 // is whether the (unmasked) addr is greater than the (unmasked) compaddr.
7
8 (boolean,boolean) DWT_AddressCompare(bits(32) addr, bits(32) compaddr, integer size,
9 integer compsize)
10 // addr must be a multiple of size. Unaligned accesses are split into smaller accesses.
11 assert Align(addr, size) == addr;
12
13 // compaddr must be a multiple of compsize
14 if Align(compaddr, compsize) != compaddr then UNPREDICTABLE;
15
16 addrmatch = (Align(addr, compsize) == Align(compaddr, size));
17 addrgreater = (UInt(addr) > UInt(compaddr));
18 return (addrmatch,addrgreater);

```

### E2.1.75 DWT\_CycCountMatch

```

1 // DWT_CycCountMatch
2 // =====
3 // Check for DWT cycle count match. This is called for each increment of
4 // DWT_CYCCNT.
5
6 DWT_CycCountMatch()
7 boolean trigger_debug_event = FALSE;
8 boolean debug_event = FALSE;
9 N = UInt(DWT_CTRL.NUMCOMP);
10 if N == 0 then return; // No comparator support
11 secure_match = IsSecure() && DWT_CTRL.CYCDISS == '1';
12 for i = 0 to N-1
13 if IsDWTConfigUnpredictable(i) then UNPREDICTABLE;
14 if DWT_FUNCTION[i].MATCH == '0001' && DWT_ValidMatch(i, secure_match)
15 && DWT_CYCCNT == DWT_COMP[i] then
16 DWT_FUNCTION[i].MATCHED = '1';
17 debug_event = DWT_FUNCTION[i].ACTION == '01';
18 trigger_debug_event = trigger_debug_event || debug_event;
19
20 // Setting the debug event if atleast one comparator matches
21 if trigger_debug_event then
22 debug_event = SetDWTDebugEvent(secure_match);
23 return;

```

### E2.1.76 DWT\_DataAddressMatch

```

1 // DWT_DataAddressMatch()
2 // =====
3 // Check for match of access at "daddr". "dsize", "read" and "NSreq" are the attributes
4 // for the access. Note that for a load or store instruction, "NSreq" is the current
5 // Security state of the PE, but this is not necessarily true for a hardware stack

```

```

6 // push/pop or vector table access. "NSreq" might not be the same as the "NSattr"
7 // attribute the PE finally uses to make the access.
8 // If comparators 'm' and 'm+1' form an Data Address Range comparator, then this function
9 // returns the range match result when N=m+1.
10
11 boolean DWT_DataAddressMatch(integer N, bits(32) daddr, integer dsize, boolean read,
12 boolean NSreq)
13 assert N < UInt(DWT_CTRL.NUMCOMP) && dsize IN {1,2,4} && Align(daddr, dsize) == daddr;
14
15 valid_match = DWT_ValidMatch(N, !NSreq);
16 valid_addr = DWT_FUNCTION[N].MATCH == 'x1xx';
17
18 if valid_match && valid_addr then
19 if N != UInt(DWT_CTRL.NUMCOMP)-1 then
20 linked_to_addr = DWT_FUNCTION[N+1].MATCH == '0111'; // Data Address Limit
21 linked_to_data = DWT_FUNCTION[N+1].MATCH == '1011'; // Linked Data Value
22 else
23 linked_to_addr = FALSE; linked_to_data = FALSE;
24
25 case DWT_FUNCTION[N].MATCH[1:0] of
26 when '00' match_lsc = TRUE; linked = FALSE;
27 when '01' match_lsc = !read; linked = FALSE;
28 when '10' match_lsc = read; linked = FALSE;
29 when '11'
30
31 case DWT_FUNCTION[N-1].MATCH[1:0] of
32 when '00' match_lsc = TRUE; linked = TRUE;
33 when '01' match_lsc = !read; linked = TRUE;
34 when '10' match_lsc = read; linked = TRUE;
35
36 if !linked_to_addr then
37 vsize = 2^UInt(DWT_FUNCTION[N].DATAVSIZE);
38 (match_eq,match_gt) = DWT_AddressCompare(daddr, DWT_COMP[N], dsize, vsize);
39
40 if linked then
41 valid_match = DWT_ValidMatch(N-1, !NSreq);
42 (lower_eq,lower_gt) = DWT_AddressCompare(daddr, DWT_COMP[N-1], dsize, 1);
43 match_addr = valid_match && (lower_eq || lower_gt) && !match_gt;
44 else
45 match_addr = match_eq;
46 else
47 match_addr = FALSE;
48
49 match = match_addr && match_lsc;
50 else
51 match = FALSE;
52
53 return match;

```

### E2.1.77 DWT\_DataMatch

```

1 // DWT_DataMatch()
2 // =====
3 // Perform varioius Data match checks for DWT
4
5 DWT_DataMatch(bits(32) daddr, integer dsize, bits(32) dvalue, boolean read, boolean NSreq)
6
7 boolean trigger_debug_event = FALSE;
8 boolean debug_event = FALSE;
9
10 if !HaveDWT() || IsZero(DWT_CTRL.NUMCOMP) then return; // No comparator
 support
11
12 for i = 0 to UInt(DWT_CTRL.NUMCOMP) - 1
13 if IsDWTConfigUnpredictable(i) then UNPREDICTABLE;
14 daddr_match = DWT_DataAddressMatch(i, daddr, dsize, read, NSreq);
15 dvalue_match = DWT_DataValueMatch(i, daddr, dvalue, dsize, read, NSreq);
16

```

```

17 // Data Address and Data Address Limit
18 if daddr_match && DWT_FUNCTION[i].MATCH == '01xx' then
19 // Data Address
20 if DWT_FUNCTION[i].MATCH != '0111' then
21 DWT_FUNCTION[i].MATCHED = '1';
22 debug_event = DWT_FUNCTION[i].ACTION == '01';
23
24 // Data Address with Data Address Limit
25 else
26 //ith comparator
27 DWT_FUNCTION[i].MATCHED = bit UNKNOWN;
28 // (i-1)th comparator
29 DWT_FUNCTION[i-1].MATCHED = '1';
30 debug_event = DWT_FUNCTION[i-1].ACTION == '01';
31
32 // Data Value and Linked Data Value
33 if dvalue_match && DWT_FUNCTION[i].MATCH == '10xx' then
34 // Data Value
35 if DWT_FUNCTION[i].MATCH != '1011' then
36 DWT_FUNCTION[i].MATCHED = '1';
37 debug_event = DWT_FUNCTION[i].ACTION == '01';
38
39 // For Linked Data Value, daddr_match will be TRUE for [i-1]
40 else
41 DWT_FUNCTION[i].MATCHED = '1';
42 debug_event = DWT_FUNCTION[i].ACTION == '01';
43
44 // Data Address with Value
45 if daddr_match && DWT_FUNCTION[i].MATCH == '11xx' then
46 DWT_FUNCTION[i].MATCHED = '1';
47 // No debug_event generated in the case of Data Address with Value
48
49 trigger_debug_event = trigger_debug_event || debug_event;
50
51 // Setting the debug event if at least one comparator matches
52 if trigger_debug_event then
53 debug_event = SetDWTDebugEvent(!NSreq);
54
55 return;

```

### E2.1.78 DWT\_DataValueMatch

```

1 // DWT_DataValueMatch()
2 // =====
3 // Check for match of access of "dvalue" at "daddr". "dsize", "read" and "NSreq"
4 // are the attributes for the access. Note that for a load or store instruction,
5 // "NSreq" is the current Security state of the PE, but this is not necessarily
6 // true for a hardware stack push/pop or vector table access. "NSreq" might not
7 // be the same as the "NSattr" attribute the PE finally uses to make the access.
8
9 boolean DWT_DataValueMatch(integer N, bits(32) daddr, bits(32) dvalue, integer dsize,
10 boolean read, boolean NSreq)
11 assert N < UInt(DWT_CTRL.NUMCOMP) && dsize IN {1,2,4} && Align(daddr,dsize) == daddr;
12
13 valid_match = DWT_ValidMatch(N, !NSreq);
14 valid_data = DWT_FUNCTION[N].MATCH[3:2] == '10';
15
16 if valid_match && valid_data then
17 case DWT_FUNCTION[N].MATCH[1:0] of
18 when '00' match_lsc = TRUE; linked = FALSE;
19 when '01' match_lsc = !read; linked = FALSE;
20 when '10' match_lsc = read; linked = FALSE;
21 when '11'
22 case DWT_FUNCTION[N-1].MATCH[1:0] of
23 when '00' match_lsc = TRUE; linked = TRUE;
24 when '01' match_lsc = !read; linked = TRUE;
25 when '10' match_lsc = read; linked = TRUE;
26

```

```

27 vsize = 2^UInt(DWT_FUNCTION[N].DATAVSZ);
28
29 // Determine which bytes of dvalue to look at in the comparison.
30 if linked then
31 dmask = '0000'; // Filled in below if there is
32 // an address match
33 if DWT_DataAddressMatch(N-1, daddr, dsize, read, NSreq) then
34 case (vsize,dsize) of
35 when (1,1) dmask[0] = '1';
36 when (1,2) dmask[UInt(DWT_COMP[N-1][0])] = '1';
37 when (1,4) dmask[UInt(DWT_COMP[N-1][1:0])] = '1';
38 when (2,2) dmask[1:0] = '11';
39 when (2,4)
40 dmask[UInt(DWT_COMP[N-1][1:0])+1:UInt(DWT_COMP[N-1][1:0])] = '11';
41 when (4,4) dmask = '1111';
42 otherwise dmask = '0000'; // vsize > dsize: no match
43
44 else
45 case dsize of
46 when 1 dmask = '0001';
47 when 2 dmask = '0011';
48 when 4 dmask = '1111';
49
50 // Split both values into byte lanes: DCBA and dcba.
51 // This function relies on the values being correctly replicated across DWT_COMP[N].
52 D = dvalue[31:24]; C = dvalue[23:16]; B = dvalue[15:8]; A = dvalue[7:0];
53 d = DWT_COMP[N][31:24]; c = DWT_COMP[N][23:16];
54 b = DWT_COMP[N][15:8]; a = DWT_COMP[N][7:0];
55
56 // Partial results
57 D_d = dmask[3] == '1' && D == d;
58 C_c = dmask[2] == '1' && C == c;
59 B_b = dmask[1] == '1' && B == b;
60 A_a = dmask[0] == '1' && A == a;
61
62 // Combined partial results
63 BA_ba = B_b && A_a;
64 DC_dc = D_d && C_c;
65 DCBA_dcba = D_d && C_c && B_b && A_a;
66
67 // Generate full results
68 case (vsize,dsize) of
69 when (1,-) match_data = D_d || C_c || B_b || A_a;
70 when (2,2), (2,4) match_data = DC_dc || BA_ba;
71 when (4,4) match_data = DCBA_dcba;
72 otherwise match_data = FALSE;
73
74 match = match_data && match_lsc;
75 else
76 match = FALSE;
77 return match;

```

## E2.1.79 DWT\_InstructionAddressMatch

```

1 // DWT_InstructionAddressMatch()
2 // =====
3 // Check for match of instruction access at "Iaddr".
4 // If comparators 'm' and 'm+1' form an Instruction Address Range comparator, then this
5 // function returns the range match when N=m+1.
6
7 boolean DWT_InstructionAddressMatch(integer N, bits(32) Iaddr)
8 assert N < UInt(DWT_CTRL.NUMCOMP) && Align(Iaddr, 2) == Iaddr;
9
10 secure_match = IsSecure();
11 valid_match = DWT_ValidMatch(N, secure_match);
12 valid_instr = DWT_FUNCTION[N].MATCH == '001x';
13
14 if valid_match && valid_instr then

```

```

15 if N != UInt(DWT_CTRL.NUMCOMP)-1 then
16 linked_to_instr = DWT_FUNCTION[N+1].MATCH == '0011';
17 else
18 linked_to_instr = FALSE;
19
20 if DWT_FUNCTION[N].MATCH == '0011' then
21 linked = TRUE;
22 else
23 linked = FALSE;
24
25 if !linked_to_instr then
26 (match_eq,match_gt) = DWT_AddressCompare(Iaddr, DWT_COMP[N], 2, 2);
27 if linked then
28 valid_match = DWT_ValidMatch(N-1, secure_match);
29 (lower_eq,lower_gt) = DWT_AddressCompare(Iaddr, DWT_COMP[N-1], 2, 2);
30 match_addr = valid_match && (lower_eq || lower_gt) && !match_gt;
31 else
32 match_addr = match_eq;
33 else
34 match_addr = FALSE;
35 match = match_addr;
36 else
37 match = FALSE;
38
39 return match;

```

### E2.1.80 DWT\_InstructionMatch

```

1 // DWT_InstructionMatch()
2 // =====
3 // Perform varioius Instruction Address checks for DWT
4
5 DWT_InstructionMatch(bits(32) Iaddr)
6
7 boolean trigger_debug_event = FALSE;
8 boolean debug_event = FALSE;
9
10 if !HaveDWT() || IsZero(DWT_CTRL.NUMCOMP) then return; // No comparator
11 support
12
13 for i = 0 to UInt(DWT_CTRL.NUMCOMP) - 1
14 if IsDWTConfigUnpredictable(i) then UNPREDICTABLE;
15 instr_addr_match = DWT_InstructionAddressMatch(i, Iaddr);
16 if instr_addr_match then
17 // Instruction Address
18 if DWT_FUNCTION[i].MATCH == '0010' then
19 DWT_FUNCTION[i].MATCHED = '1';
20 debug_event = DWT_FUNCTION[i].ACTION == '01';
21
22 // Instruction Address Limit
23 elseif DWT_FUNCTION[i].MATCH == '0011' then
24 DWT_FUNCTION[i].MATCHED = bit UNKNOWN;
25 DWT_FUNCTION[i-1].MATCHED = '1';
26 debug_event = DWT_FUNCTION[i-1].ACTION == '01';
27
28 trigger_debug_event = trigger_debug_event || debug_event;
29
30 if trigger_debug_event then
31 debug_event = SetDWTDebugEvent(IsSecure());
32 return;

```

### E2.1.81 DWT\_ValidMatch

```

1 // DWT_ValidMatch()
2 // =====
3 // Returns TRUE if this match is permitted by the current authentication controls, FALSE
4 // otherwise.

```

```

4
5 boolean DWT_ValidMatch(integer N, boolean secure_match)
6 if !HaveSecurityExt() then assert !secure_match;
7
8 // Check for disabled
9 if ((secure_match && !SecureNoninvasiveDebugAllowed()) || !NoninvasiveDebugAllowed()) ||
10 DEMCR.TRCENA == '0' || DWT_FUNCTION[N].MATCH == '0000' then
11 return FALSE;
12
13 // Check for Debug event
14 if DWT_FUNCTION[N].ACTION == '01' then
15 hlt_en = CanHaltOnEvent(secure_match);
16 // Ignore priority when checking whether DebugMonitor activates DWT matches
17 mon_en = HaveDebugMonitor() && CanPendMonitorOnEvent(secure_match, FALSE);
18 return (hlt_en || mon_en);
19 else
20 // Otherwise trace or trigger event
21 return !secure_match || SecureNoninvasiveDebugAllowed();

```

### E2.1.82 EndOfInstruction

```

1 // EndOfInstruction
2 // =====
3 // Terminates the processing of current instruction.
4
5 EndOfInstruction();

```

### E2.1.83 EventRegistered

```

1 // EventRegistered
2 // =====
3 // Returns TRUE if PE Event Register is set to 1 and FALSE otherwise.
4
5 boolean EventRegistered();

```

### E2.1.84 ExceptionActiveBitCount

```

1 // ExceptionActiveBitCount()
2 // =====
3
4 integer ExceptionActiveBitCount()
5 integer count = 0;
6 for i = 0 to MaxExceptionNum()
7 for j = 0 to 1
8 if IsActiveForState(i, j == 0) then
9 count = count + 1;
10 return count;

```

### E2.1.85 ExceptionDetails

```

1 // ExceptionDetails()
2 // =====
3
4 (boolean, boolean) ExceptionDetails(integer exception, boolean isSecure, boolean
5 isSynchronous)
6 // Is the exception subject to escalation
7 case exception of
8 when HardFault
9 termInst = TRUE;
10 enabled = TRUE;
11 canEscalate = TRUE;
12 when MemManage
13 termInst = TRUE;
14 if HaveMainExt() then

```

```

14 val = if isSecure then SHCSR_S else SHCSR_NS;
15 enabled = val.MEMFAULTENA == '1';
16 else
17 enabled = FALSE;
18 canEscalate = TRUE;
19 when BusFault
20 termInst = isSynchronous;
21 enabled = if HaveMainExt()
22 then SHCSR_S.BUSFAULTENA == '1' else FALSE;
23 // Async BusFaults only escalate if they are disabled
24 canEscalate = termInst || !enabled;
25 when UsageFault
26 termInst = TRUE;
27 if HaveMainExt() then
28 val = if isSecure then SHCSR_S else SHCSR_NS;
29 enabled = val.USGFAULTENA == '1';
30 else
31 enabled = FALSE;
32 canEscalate = TRUE;
33 when SecureFault
34 termInst = TRUE;
35 enabled = if HaveMainExt()
36 then SHCSR_S.SECUREFAULTENA == '1' else FALSE;
37 canEscalate = TRUE;
38 when SVCcall
39 termInst = FALSE;
40 enabled = TRUE;
41 canEscalate = TRUE;
42 when DebugMonitor
43 termInst = TRUE;
44 enabled = if HaveMainExt()
45 then DEMCR.MON_EN == '1' else FALSE;
46 canEscalate = FALSE; // TRUE if fault caused by BKPT instruction
47 otherwise
48 termInst = FALSE;
49 canEscalate = FALSE;
50
51 // If the fault can escalate then check if exception can be taken immediately, or whether
52 // it should escalate.
53 // NOTE: In some cases (for example faults during lazy floating-point state preservation)
54 // the priority comparison below is ignored and the decision to escalate or not is
55 // based on other factors.
56 escalateToHf = FALSE;
57 if canEscalate then
58 execPri = ExecutionPriority();
59 excePri = ExceptionPriority(exception, isSecure, TRUE);
60 if (excePri >= execPri) || !enabled then
61 escalateToHf = TRUE;
62
63 return (escalateToHf, termInst);

```

### E2.1.86 ExceptionEnabled

```

1 // ExceptionEnabled()
2 // =====
3 // Checks whether the given exception is enabled.
4
5 boolean ExceptionEnabled(integer exception, boolean secure);

```

### E2.1.87 ExceptionEntry

```

1 // ExceptionEntry()
2 // =====
3 // Exception entry is modified according to the behavior of a derived
4 // exception, see DerivedLateArrival() also.
5
6 ExcInfo ExceptionEntry(integer exceptionType, boolean toSecure, boolean instExecOk)

```



```

7
8 // PushStack() can abandon memory accesses if a fault occurs during the stacking
9 // sequence.
10 exc = PushStack(toSecure, instExecOk);
11 if exc.fault == NoFault then
12 exc = ExceptionTaken(exceptionType, FALSE, toSecure, FALSE);
13 return exc;

```

## E2.1.88 ExceptionPriority

```

1 // ExceptionPriority()
2 // =====
3
4 integer ExceptionPriority(integer n, boolean isSecure, boolean groupPri)
5 if HaveMainExt() then
6 assert n >= 1 && n <= 511;
7 else
8 assert n >= 1 && n <= 48;
9
10 if n == Reset then // Reset
11 result = -4;
12 elseif n == NMI then // NMI
13 result = -2;
14 elseif n == HardFault then //HardFault
15 if isSecure && AIRCR.BFHFNMINS == '1' then
16 result = -3;
17 else
18 result = -1;
19 elseif HaveMainExt() && n == MemManage then // MemManage
20 result = UInt(if isSecure then SHPR1_S.PRI_4 else SHPR1_NS.PRI_4);
21 elseif HaveMainExt() && n == BusFault then // BusFault
22 result = UInt(SHPR1_S.PRI_5);
23 elseif HaveMainExt() && n == UsageFault then // UsageFault
24 result = UInt(if isSecure then SHPR1_S.PRI_6 else SHPR1_NS.PRI_6);
25 elseif HaveMainExt() && n == SecureFault then // SecureFault
26 result = UInt(SHPR1_S.PRI_7);
27 elseif n == SVCcall then // SVCcall
28 result = UInt(if isSecure then SHPR2_S.PRI_11 else SHPR2_NS.PRI_11);
29 elseif HaveMainExt() && n == DebugMonitor then // DebugMonitor
30 result = UInt(SHPR3_S.PRI_12);
31 elseif n == PendSV then // PendSV
32 result = UInt(if isSecure then SHPR3_S.PRI_14 else SHPR3_NS.PRI_14);
33 elseif n == SysTick // SysTick
34 && ((HaveSysTick() == 2) ||
35 (HaveSysTick() == 1 && ((ICSR_S.STTNS == '0') == isSecure))) then
36 result = UInt(if isSecure then SHPR3_S.PRI_15 else SHPR3_NS.PRI_15);
37 elseif n >= 16 then // External interrupt (n-16)
38 r = (n - 16) DIV 4;
39 v = n MOD 4;
40 result = UInt(NVIC_IPR[r][v*8+7:v*8]);
41 else // Reserved exceptions
42 result = 256;
43
44 // Negative priorities (ie Reset, NMI, and HardFault) are not effected by
45 // PRIGROUP or PRIS
46 if result >= 0 then
47 // Include the PRIGROUP effect
48 if HaveMainExt() && groupPri then
49 integer subgroupshift;
50 if isSecure then
51 subgroupshift = UInt(AIRCR_S.PRIGROUP);
52 else
53 subgroupshift = UInt(AIRCR_NS.PRIGROUP);
54 integer groupvalue = 2 << subgroupshift;
55 integer subgroupvalue = result MOD groupvalue;
56 result = result - subgroupvalue;
57
58 PriSNsPri = RestrictedNSPri();

```

```

59 if (AIRCR_S.PRIS == '1') && !isSecure then
60 result = (result >> 1) + PriSNSPri;
61
62 return result;

```

## E2.1.89 ExceptionReturn

```

1 // ExceptionReturn()
2 // =====
3
4 (ExcInfo, EXC_RETURN_Type) ExceptionReturn(EXC_RETURN_Type excReturn)
5 integer returningExceptionNumber = UInt(IPSR.Exception);
6
7 (exc, excReturn) = ValidateExceptionReturn(excReturn, returningExceptionNumber);
8 if exc.fault != NoFault then
9 return (exc, excReturn);
10
11 if HaveSecurityExt() then
12 excSecure = excReturn.ES == '1';
13 retToSecure = excReturn.S == '1';
14 else
15 excSecure = FALSE;
16 retToSecure = FALSE;
17
18 // Restore SPSEL for the Security state we are returning from.
19 if excSecure then
20 CONTROL_S.SPSEL = excReturn.SPSEL;
21 else
22 CONTROL_NS.SPSEL = excReturn.SPSEL;
23
24 targetDomainSecure = excReturn.ES == '1';
25 DeActivate(returningExceptionNumber, targetDomainSecure);
26
27 // If requested, clear the scratch FP values left in the caller saved
28 // registers before returning/tail chaining.
29 if HaveFPExt() && FPCCR.CLRONRET == '1' && CONTROL.FPCA == '1' then
30 if FPCCR_S.LSPACT == '1' then
31 SFSR.LSERR = '1';
32 exc = CreateException(SecureFault, TRUE, TRUE);
33 return (exc, excReturn);
34 else
35 for i = 0 to 15
36 S[i] = Zeros();
37 FPSCR = Zeros();
38
39 // If TailChaining is supported, check if there is a pending exception with
40 // sufficient priority to be taken now. This check is done after the
41 // previous exception is deactivated so the priority of the previous
42 // exception doesn't mask any pending exceptions.
43 // The position of TailChain() within this function is the earliest point
44 // at which an tailchain is architecturally visible. Tail-chaining from a
45 // later point is permissible.
46 if boolean IMPLEMENTATION_DEFINED "Tail chaining supported" then
47 (takeException, exception, excIsSecure) = PendingExceptionDetails();
48 if takeException then
49 exc = TailChain(exception, excIsSecure, excReturn);
50 return (exc, excReturn);
51
52 // Return to the background Security state
53 if HaveSecurityExt() then
54 CurrentState = if retToSecure
55 then SecurityState_Secure else SecurityState_NonSecure;
56
57 // Sleep-on-exit performs equivalent behavior to the WFI instruction.
58 // The position of SleepOnExit() within this function is the earliest point
59 // at which it can be performed. Performing SleepOnExit from a later point
60 // is permissible.
61 if (excReturn.Mode == '1' && SCR.SLEEPONEXIT == '1' &&

```

```

62 ExceptionActiveBitCount() == 0) then
63 SleepOnExit(); // IMPLEMENTATION DEFINED
64
65 // Pop the stack and raise any exceptions that are generated
66 exc = PopStack(excReturn);
67 if exc.fault == NoFault then
68 ClearExclusiveLocal(ProcessorID());
69 SetEventRegister(); // See WFE instruction for more details
70 InstructionSynchronizationBarrier('1111');
71
72 return (exc, excReturn);

```

## E2.1.90 ExceptionTaken

```

1 // ExceptionTaken()
2 // =====
3
4 ExcInfo ExceptionTaken(integer exceptionNumber, boolean doTailChain,
5 boolean excIsSecure, boolean ignStackFaults)
6 assert(HaveSecurityExt() || !excIsSecure);
7
8 // If the background code was running in the Secure state that are some
9 // additional steps that might need to be taken to protect the callee saved
10 // registers
11 exc = DefaultExcInfo();
12 if HaveSecurityExt() && LR[6] == '1' then
13 if excIsSecure then // Transitioning to Secure
14 // If tail chaining is from Non-secure to Secure, then the callee registers
15 // are already on stack. Set excReturn.DCRS accordingly
16 if doTailChain && LR[0] == '0' then
17 LR[5] = '0';
18 else // Transitioning to Non-secure
19 // If the callee registers aren't already on the stack push them now
20 if LR[5] == '1' && !(doTailChain && LR[0] == '0') then
21 exc = PushCalleeStack(doTailChain);
22 // Going to Non-secure exception. Set excReturn.DCRS to default
23 // value
24 LR[5] = '1';
25
26 // Finalise excReturn value
27 if excIsSecure then
28 LR[2] = CONTROL_S.SPSEL;
29 LR[0] = '1';
30 else
31 LR[2] = CONTROL_NS.SPSEL;
32 LR[0] = '0';
33
34 // Register clearing
35 // Caller saved registers: These registers are cleared if exception targets
36 // the Non-secure state, otherwise they are UNKNOWN. NOTE: The original
37 // values were pushed to the stack.
38 callerRegValue = if !HaveSecurityExt() || excIsSecure then bits(32) UNKNOWN else
39 Zeros(32);
40 for n = 0 to 3
41 R[n] = callerRegValue;
42 R[12] = callerRegValue;
43 EAPSR = callerRegValue;
44 // Callee saved registers: If the background code was in the Secure state
45 // these registers are cleared if the exception targets the Non-secure state,
46 // and UNKNOWN if it targets the Secure state and the registers have been
47 // pushed to the stack (as indicated by EXC_RETURN.DCRS).
48 //
49 // NOTE: Callee saved registers are preserved if the background code is
50 // Non-secure, of when the exception is Secure and the values have not
51 // been pushed to the stack.
52 if HaveSecurityExt() && LR[6] == '1' then
53 if excIsSecure then
54 if LR[5] == '0' then

```

```

54 for n = 4 to 11
55 R[n] = bits(32) UNKNOWN;
56 else
57 for n = 4 to 11
58 R[n] = Zeros();
59
60 // If no errors so far (or errors that can be ignored) load the vector address
61 if exc.fault == NoFault || ignStackFaults then
62 (exc, start) = Vector[exceptionNumber, excIsSecure];
63
64 // The state or mode of processor is not updated if an exception is raised
65 // during the entry sequence.
66 if exc.fault == NoFault then
67 ActivateException(exceptionNumber, excIsSecure);
68 SCS_UpdateStatusRegs();
69 ClearExclusiveLocal(ProcessorID());
70 SetEventRegister(); // See WFE instruction for details
71 InstructionSynchronizationBarrier('1111');
72 // Start execution of handler
73 EPSR.T = start[0];
74 // If EPSR.T == 0 then an exception is taken on the next
75 // instruction: UsageFault('Invalid State') if the Main Extension is
76 // implemented; HardFault otherwise
77 BranchTo(start[31:1]:'0');
78 else
79 exc.inExcTaken = TRUE;
80 return exc;

```

### E2.1.91 ExceptionTargetsSecure

```

1 // ExceptionTargetsSecure()
2 // =====
3
4 // Determine the default Security state an exception is expected to target if the
5 // exception is not forced to a specific domain.
6
7 boolean ExceptionTargetsSecure(integer exceptionNumber, boolean isSecure)
8 if !HaveSecurityExt() then
9 return FALSE;
10
11 boolean targetSecure = FALSE;
12 case exceptionNumber of
13 when NMI
14 targetSecure = AIRCR.BFHFNMIN == '0';
15
16 when HardFault
17 targetSecure = AIRCR.BFHFNMIN == '0' || isSecure;
18
19 when MemManage
20 targetSecure = isSecure;
21
22 when BusFault
23 targetSecure = AIRCR.BFHFNMIN == '0';
24
25 when UsageFault
26 targetSecure = isSecure;
27
28 when SecureFault
29 // SecureFault always targets Secure state.
30 targetSecure = TRUE;
31
32 when SVCall
33 targetSecure = isSecure;
34
35 when DebugMonitor
36 targetSecure = DEMCR.SDME == '1';
37
38 when PendSV

```

```

39 // This state should be unreachable as PendSV is a banked interrupt
40 // and it is directly pending for the correct security state, so this
41 // function is not called for this exception.
42 assert FALSE;
43
44 when SysTick
45 if HaveSysTick() != 1 then
46 // If there is a SysTick for each domain, then the exception
47 // targets the domain associated with the SysTick instance that
48 // raised the exception.
49 // This state should be unreachable as SysTick exception is banked
50 // and it is directly pending for the correct security state. This
51 // function can only be called when 1 SysTick is implemented.
52 assert FALSE;
53 else
54 // SysTick target state is configurable
55 targetSecure = ICSR_S.STTNS == '0';
56
57 otherwise
58 if exceptionNumber >= 16 then
59 // Interrupts target the state defined by the NVIC_ITNS register
60 targetSecure = NVIC_ITNS[exceptionNumber - 16] == '0';
61
62 return targetSecure;

```

### E2.1.92 ExcInfo

```

1 // Exception information
2
3 type ExcInfo is (
4 integer fault, // The ID of the resulting fault, or NoFault (ie 0)
5 // if no fault occurred
6 integer origFault, // The ID of the original fault raised before
7 // escalation is considered.
8 boolean isSecure, // TRUE if the fault targets the Secure state.
9 boolean origFaultIsSecure, // TRUE if the original fault raised targeted
10 // Secure state
11 boolean isTerminal, // Set to TRUE for derived faults (eg exception on
12 // exception entry) that prevent the original
13 // exception being entered (eg a BusFault whilst
14 // fetching the exception vector address).
15 boolean inExcTaken, // TRUE if the exception occurred during ExceptionTaken()
16 // This is used to determine if the LR update and the
17 // callee stacking operations have been performed, and
18 // therefore whether the derived exception should be
19 // treated as a tail chain.
20 boolean lockup, // Set to TRUE if the exception should cause a lockup.
21 boolean termInst // Set to TRUE if the exception should cause the
22 // instruction to be terminated.
23)

```

### E2.1.93 ExclusiveMonitorsPass

```

1 // ExclusiveMonitorsPass()
2 // =====
3
4 boolean ExclusiveMonitorsPass(bits(32) address, integer size)
5
6 // It is IMPLEMENTATION DEFINED whether the detection of memory aborts happens
7 // before or after the check on the local Exclusive Monitor. As a result a failure
8 // of the local monitor can occur on some implementations even if the memory
9 // access would give a memory abort.
10
11 if address != Align(address, size) then
12 UFSR.UNALIGNED = '1';
13 excInfo = CreateException(UsageFault, FALSE, boolean UNKNOWN);
14 else

```

```

15 (excInfo, memaddrdesc) = ValidateAddress(address, AccType_NORMAL,
16 FindPriv(), IsSecure(), TRUE, TRUE);
17 HandleException(excInfo);
18
19 passed = IsExclusiveLocal(memaddrdesc.address, ProcessorID(), size);
20 if memaddrdesc.memattrs.shareable then
21 passed = passed && IsExclusiveGlobal(memaddrdesc.address, ProcessorID(), size);
22 if passed then
23 ClearExclusiveLocal(ProcessorID());
24 return passed;

```

### E2.1.94 ExecuteCPCheck

```

1 // ExecuteCPCheck()
2 // =====
3
4 ExecuteCPCheck(integer cp)
5 // Check access to coprocessor is enabled
6 excInfo = CheckCPEEnabled(cp);
7 HandleException(excInfo);

```

### E2.1.95 ExecuteFPCheck

```

1 // ExecuteFPCheck()
2 // =====
3
4 ExecuteFPCheck()
5 // If FP lazy context save is enabled then save state
6 if FPCCR_S.S == '1' then
7 lspact = FPCCR_S.LSPACT;
8 else
9 lspact = FPCCR_NS.LSPACT;
10 if lspact == '1' then
11 PreserveFPState();
12
13 // Update the ownership of the FP context
14 FPCCR_S.S = if IsSecure() then '1' else '0';
15
16 // Update CONTROL.FPCA, and create new FP context
17 // if this has been enabled by setting FPCCR.ASPEN to 1
18 if FPCCR.ASPEN == '1' &&
19 (CONTROL.FPCA == '0' || (IsSecure() && CONTROL_S.SFPA == '0')) then
20 FPSCR = FPDSCR[31:0];
21 CONTROL.FPCA = '1';
22 if IsSecure() then
23 CONTROL_S.SFPA = '1';
24 return;

```

### E2.1.96 ExecutionPriority

```

1 // ExecutionPriority()
2 // =====
3 // Determine the current execution priority
4
5 integer ExecutionPriority()
6
7 boostedpri = HighestPri(); // Priority influence of BASEPRI, PRIMASK and FAULTMASK
8
9 // Calculate boosted priority effect due to BASEPRI for both Security states
10 PriSnsPri = RestrictedNSPri();
11 if HaveMainExt() then
12 if UInt(BASEPRI_NS[7:0]) != 0 then
13 basepri = UInt(BASEPRI_NS[7:0]);
14 // Include the PRIGROUP effect
15 subgroupshift = UInt(AIRCR_NS.PRIGROUP);

```

```

16 groupvalue = 2 << subgroupshift;
17 subgroupvalue = basepri MOD groupvalue;
18 boostedpri = basepri - subgroupvalue;
19 if AIRCR_S.PRIS == '1' then
20 boostedpri = (boostedpri >> 1) + PriSNsPri;
21
22 if UInt(BASEPRI_S[7:0]) != 0 then
23 basepri = UInt(BASEPRI_S[7:0]);
24 // Include the PRIGROUP effect
25 subgroupshift = UInt(AIRCR_S.PRIGROUP);
26 groupvalue = 2 << subgroupshift;
27 subgroupvalue = basepri MOD groupvalue;
28 basepri = basepri - subgroupvalue;
29 if boostedpri > basepri then
30 boostedpri = basepri;
31
32 // Calculate boosted priority effect due to PRIMASK for both Security states
33 if PRIMASK_NS.PM == '1' then
34 if AIRCR_S.PRIS == '0' then
35 boostedpri = 0;
36 else
37 if boostedpri > PriSNsPri then
38 boostedpri = PriSNsPri;
39
40 if PRIMASK_S.PM == '1' then
41 boostedpri = 0;
42
43 // Calculate boosted priority effect due to FAULTMASK for both Security states
44 if HaveMainExt() then
45 if FAULTMASK_NS.FM == '1' then
46 if AIRCR.BFHFNMINs == '0' then
47 if AIRCR_S.PRIS == '0' then
48 boostedpri = 0;
49 else
50 if boostedpri > PriSNsPri then
51 boostedpri = PriSNsPri;
52 else
53 boostedpri = -1;
54
55 if FAULTMASK_S.FM == '1' then
56 boostedpri = if AIRCR.BFHFNMINs == '0' then -1 else -3;
57
58 // Finally calculate the resultant priority after boosting
59 rawExecPri = RawExecutionPriority();
60 if boostedpri < rawExecPri then
61 priority = boostedpri;
62 else
63 priority = rawExecPri;
64
65 return priority;

```

### E2.1.97 ExternalInvasiveDebugEnabled

```

1 // ExternalInvasiveDebugEnabled()
2 // =====
3 // Return TRUE if Halting debug is enabled by the IMPLEMENTATION DEFINED authentication
4 // interface.
5
6 boolean ExternalInvasiveDebugEnabled()
7 // In the recommended interface, ExternalInvasiveDebugEnabled returns the state of
8 // the DBGGEN signal.
9 return DBGGEN == HIGH;

```

### E2.1.98 ExternalNoninvasiveDebugEnabled

```

1 // ExternalNoninvasiveDebugEnabled()
2 // =====

```

```

3 // Return TRUE if non-invasive debug is enabled by the IMPLEMENTATION DEFINED authentication
4 // interface.
5
6 boolean ExternalNoninvasiveDebugEnabled()
7 // In the recommended interface, ExternalNoninvasiveDebugEnabled returns the state of
8 // the (DBGEN OR NIDEN) signal.
9 return ExternalInvasiveDebugEnabled() || NIDEN == HIGH;

```

### E2.1.99 ExternalSecureInvasiveDebugEnabled

```

1 // ExternalSecureInvasiveDebugEnabled()
2 // =====
3 // Return TRUE if Secure Halting debug is enabled by the IMPLEMENTATION DEFINED
4 // authentication
5 // interface.
6
7 boolean ExternalSecureInvasiveDebugEnabled()
8 // In the recommended interface, ExternalSecureInvasiveDebugEnabled returns the state
9 // of the (DBGEN AND SPIDEN) signal.
10 return ExternalInvasiveDebugEnabled() && SPIDEN == HIGH;

```

### E2.1.100 ExternalSecureNoninvasiveDebugEnabled

```

1 // ExternalSecureNoninvasiveDebugEnabled()
2 // =====
3 // Return TRUE if Secure non-invasive debug is enabled by the IMPLEMENTATION DEFINED
4 // authentication
5 // interface.
6
7 boolean ExternalSecureNoninvasiveDebugEnabled()
8 // In the recommended interface, ExternalSecureNoninvasiveDebugEnabled returns the
9 // state of the (DBGEN OR NIDEN) AND (SPIDEN OR SPNIDEN) signal.
10 return ExternalNoninvasiveDebugEnabled() && (SPIDEN == HIGH || SPNIDEN == HIGH);

```

### E2.1.101 ExternalSecureSelfHostedDebugEnabled

```

1 // ExternalSecureSelfHostedDebugEnabled()
2 // =====
3 // Return TRUE if Secure self-hosted debug is enabled by the IMPLEMENTATION DEFINED
4 // authentication
5 // interface.
6
7 boolean ExternalSecureSelfHostedDebugEnabled()
8 // In the recommended interface, ExternalSecureSelfHostedDebugEnabled returns the state
9 // of the (DBGEN AND SPIDEN) signal.
10 return DBGEN == HIGH && SPIDEN == HIGH;

```

### E2.1.102 FaultNumbers

```

1 // Fault Numbers
2 // =====
3
4 // The fault numbers are a subset of ExceptionNumber and can be one of the
5 // following values:
6 constant integer NoFault = 0;
7 constant integer Reset = 1;
8 constant integer NMI = 2;
9 constant integer HardFault = 3;
10 constant integer MemManage = 4;
11 constant integer BusFault = 5;
12 constant integer UsageFault = 6;
13 constant integer SecureFault = 7;
14 constant integer SVCcall = 11;
15 constant integer DebugMonitor = 12;

```



```

16 constant integer PendSV = 14;
17 constant integer SysTick = 15;

```

### E2.1.103 FetchInstr

```

1 // FetchInstr()
2 // =====
3
4 (bits(32), boolean) FetchInstr(bits(32) addr)
5 // NOTE: It is CONSTRAINED UNPREDICTABLE whether otherwise valid sequential
6 // instruction fetches that cross from Non-secure to Secure memory
7 // generate a INVEP SecureFault, or transition normally.
8 sgOpcode = 0xE97FE97F[31:0];
9
10 hw1Attr = SecurityCheck(addr, TRUE, IsSecure());
11 // Fetch the a T16 instruction, or the first half of a T32.
12 hw1Instr = MemI[addr];
13
14 // If the T bit is clear then the instruction can't be decoded
15 if EPSR.T == '0' then
16 // Attempted NS->S domain crossings with the T bit clear raise an INVEP
17 // SecureFault
18 if !IsSecure() && !hw1Attr.ns then
19 SFSR.INVEP = '1';
20 excInfo = CreateException(SecureFault, TRUE, TRUE);
21 else
22 UFSR.INVSTATE = '1';
23 excInfo = CreateException(UsageFault, FALSE, boolean UNKNOWN);
24 HandleException(excInfo);
25
26 // Implementations are permitted to terminate the fetch process early if a
27 // domain crossing is being attempted and the first 16bits of the opcode
28 // isn't the first part of the SG instruction.
29 if boolean IMPLEMENTATION_DEFINED "Early SG check" then
30 if !IsSecure() && !hw1Attr.ns && (hw1Instr != sgOpcode[31:16]) then
31 SFSR.INVEP = '1';
32 excInfo = CreateException(SecureFault, TRUE, TRUE);
33 HandleException(excInfo);
34
35 // NOTE: Implementations are also permitted to terminate the fetch process
36 // at this point with an UNDEFINSTR UsageFault if the first 16bit is
37 // an undefined T32 prefix.
38
39 // If the data fetched is the top half of a T32 instruction fetch the bottom
40 // 16 bits
41 isT16 = UInt(hw1Instr[15:11]) < UInt('11101');
42 if isT16 then
43 instr = Zeros(16) : hw1Instr;
44 else
45 hw2Attr = SecurityCheck(addr+2, TRUE, IsSecure());
46 // The following test covers 2 possible fault conditions:-
47 // 1) NS code branching to a T32 instruction where the first half is in
48 // NS memory, and the second half is in S memory.
49 // 2) NS code branching to a T32 instruction in S & NSC memory, but
50 // where the second half of the instruction is in NS memory.
51 if !IsSecure() && (hw1Attr.ns != hw2Attr.ns) then
52 SFSR.INVEP = '1';
53 excInfo = CreateException(SecureFault, TRUE, TRUE);
54 HandleException(excInfo);
55
56 // Fetch the second half of T32 instruction
57 instr = hw1Instr : MemI[addr+2];
58
59 // Raise a fault if an otherwise valid NS->S transition that doesn't land on
60 // an SG instruction.
61 if !IsSecure() && !hw1Attr.ns && (instr != sgOpcode) then
62 SFSR.INVEP = '1';
63 excInfo = CreateException(SecureFault, TRUE, TRUE);

```

```

64 HandleException(excInfo);
65 return (instr, isT16);

```

### E2.1.104 FindPriv

```

1 // FindPriv()
2 // =====
3
4 boolean FindPriv()
5 return CurrentModeIsPrivileged();

```

### E2.1.105 FixedToFP

```

1 // FixedToFP()
2 // =====
3
4 bits(N) FixedToFP(bits(M) operand, integer N, integer fraction_bits, boolean unsigned,
5 boolean round_to_nearest, boolean fpscr_controlled)
6 assert N IN {32,64};
7 fpscr_val = if fpscr_controlled then FPSCR else StandardFPSCRValue();
8 if round_to_nearest then fpscr_val[23:22] = '00';
9 int_operand = if unsigned then UInt(operand) else SInt(operand);
10 real_operand = Real(int_operand) / 2.0^fraction_bits;
11 if real_operand == 0.0 then
12 result = FPZero('0', N);
13 else
14 result = FPRound(real_operand, N, fpscr_val);
15 return result;

```

### E2.1.106 FPAbs

```

1 // FPAbs()
2 // =====
3
4 bits(N) FPAbs(bits(N) operand)
5 assert N IN {32,64};
6 return '0' : operand[N-2:0];

```

### E2.1.107 FPAdd

```

1 // FPAdd()
2 // =====
3
4 bits(N) FPAdd(bits(N) op1, bits(N) op2, boolean fpscr_controlled)
5 assert N IN {32,64};
6 fpscr_val = if fpscr_controlled then FPSCR else StandardFPSCRValue();
7 (type1,sign1,value1) = FPUnpack(op1, fpscr_val);
8 (type2,sign2,value2) = FPUnpack(op2, fpscr_val);
9 (done,result) = FPProcessNaNs(type1, type2, op1, op2, fpscr_val);
10 if !done then
11 inf1 = (type1 == FPType_Infinity); inf2 = (type2 == FPType_Infinity);
12 zero1 = (type1 == FPType_Zero); zero2 = (type2 == FPType_Zero);
13 if inf1 && inf2 && sign1 == NOT(sign2) then
14 result = FPDefaultNaN(N);
15 FPProcessException(FPExc_InvalidOp, fpscr_val);
16 elsif (inf1 && sign1 == '0') || (inf2 && sign2 == '0') then
17 result = FPInfinity('0', N);
18 elsif (inf1 && sign1 == '1') || (inf2 && sign2 == '1') then
19 result = FPInfinity('1', N);
20 elsif zero1 && zero2 && sign1 == sign2 then
21 result = FPZero(sign1, N);
22 else
23 result_value = value1 + value2;

```

```

24 if result_value == 0.0 then // Sign of exact zero result depends on rounding
25 mode
26 result_sign = if fpscr_val[23:22] == '10' then '1' else '0';
27 result = FPZero(result_sign, N);
28 else
29 result = FPRound(result_value, N, fpscr_val);
30 return result;

```

### E2.1.108 FPB\_BreakpointMatch

```

1 // FPB_BreakpointMatch()
2 // =====
3 // Generates a debug event based on FP Breakpoint Match
4
5 FPB_BreakpointMatch()
6 i = GenerateDebugEventResponse();

```

### E2.1.109 FPB\_CheckBreakPoint

```

1 // FPB_CheckBreakPoint
2 // =====
3 // Check for Flash Patch Break point
4
5 boolean FPB_CheckBreakPoint(bits(32) iaddr, integer size, boolean is_ifetch, boolean
6 is_secure)
7
8 match = FPB_CheckMatchAddress(iaddr);
9 if !match && size == 4 && FPB_CheckMatchAddress(iaddr + 2) then
10 match = ConstrainUnpredictableBool(Unpredictable_FPBreakpoint);
11 return match;

```

### E2.1.110 FPB\_CheckMatchAddress

```

1 // FPB_CheckMatchAddress
2 // =====
3 // Flash Patch breakpoint instruction address comparison
4
5 boolean FPB_CheckMatchAddress(bits(32) iaddr)
6
7 if FP_CTRL.ENABLE == '0' then return FALSE; // FPB not enabled
8
9 // Instruction Comparator.
10 num_addr_cmp = UInt(FP_CTRL.NUM_CODE);
11 if num_addr_cmp == 0 then return FALSE; // No comparator support
12
13 for N = 0 to (num_addr_cmp - 1)
14 if FP_COMP[N].BE == '1' then // Breakpoint enabled
15 if iaddr[31:1] == FP_COMP[N].BPADDR then
16 return TRUE;
17
18 return FALSE;

```

### E2.1.111 FPCompare

```

1 // FPCompare()
2 // =====
3
4 (bit, bit, bit, bit) FPCompare(bits(N) op1, bits(N) op2, boolean quiet_nan_exc,
5 boolean fpscr_controlled)
6
7 assert N IN {32, 64};
8 fpscr_val = if fpscr_controlled then FPSCR else StandardFPSCRValue();
9 (type1, -, value1) = FPUnpack(op1, fpscr_val);
10 (type2, -, value2) = FPUnpack(op2, fpscr_val);

```

```

10 if type1==FPType_SNaN || type1==FPType_QNaN || type2==FPType_SNaN || type2==FPType_QNaN
11 then
12 result = ('0','0','1','1');
13 if type1==FPType_SNaN || type2==FPType_SNaN || quiet_nan_exc then
14 FPProcessException(FPExc_InvalidOp, fpscr_val);
15 else
16 // All non-NaN cases can be evaluated on the values produced by FPUnpack()
17 if value1 == value2 then
18 result = ('0','1','1','0');
19 elseif value1 < value2 then
20 result = ('1','0','0','0');
21 else // value1 > value2
22 result = ('0','0','1','0');
23 return result;

```

### E2.1.112 FPDefaultNaN

```

1 // FPDefaultNaN()
2 // =====
3
4 bits(N) FPDefaultNaN(integer N)
5 assert N IN {16,32,64};
6 integer E = if N == 16 then 5 elseif N == 32 then 8 else 11;
7 constant integer F = N - E - 1;
8 bits(E) exp;
9 bits(F) frac;
10 sign = '0';
11 exp = Ones(E);
12 frac = '1':Zeros(F-1);
13 return sign : exp : frac;

```

### E2.1.113 FPDiv

```

1 // FPDiv()
2 // =====
3
4 bits(N) FPDiv(bits(N) op1, bits(N) op2, boolean fpscr_controlled)
5 assert N IN {32,64};
6 fpscr_val = if fpscr_controlled then FPSCR else StandardFPSCRValue();
7 (fp_type1,sign1,value1) = FPUnpack(op1, fpscr_val);
8 (fp_type2,sign2,value2) = FPUnpack(op2, fpscr_val);
9 (done,result) = FPProcessNaNs(fp_type1, fp_type2, op1, op2, fpscr_val);
10 if !done then
11 inf1 = (fp_type1 == FPType_Infinity); inf2 = (fp_type2 == FPType_Infinity);
12 zero1 = (fp_type1 == FPType_Zero); zero2 = (fp_type2 == FPType_Zero);
13 if (inf1 && inf2) || (zero1 && zero2) then
14 result = FPDefaultNaN(N);
15 FPProcessException(FPExc_InvalidOp, fpscr_val);
16 elseif inf1 || zero2 then
17 result_sign = if sign1 == sign2 then '0' else '1';
18 result = FPInfinity(result_sign, N);
19 if !inf1 then FPProcessException(FPExc_DivideByZero, fpscr_val);
20 elseif zero1 || inf2 then
21 result_sign = if sign1 == sign2 then '0' else '1';
22 result = FPZero(result_sign, N);
23 else
24 result = FPRound(value1/value2, N, fpscr_val);
25 return result;

```

### E2.1.114 FPDoubleToHalf

```

1 // FPDoubleToHalf()
2 // =====
3 bits(16) FPDoubleToHalf(bits(64) operand, boolean fpscr_controlled)
4 fpscr_val = if fpscr_controlled then FPSCR else StandardFPSCRValue();

```

```

5 (fp_type,sign,value) = FPUnpack(operand, fpscr_val);
6 if fp_type == FPType_SNaN || fp_type == FPType_QNaN then
7 if fpscr_val[26] == '1' then // AH bit set
8 result = FPZero(sign, 16);
9 elsif fpscr_val[25] == '1' then // DN bit set
10 result = FPDefaultNaN(16);
11 else
12 result = sign : '11111 1' : operand[50:42];
13 if fp_type == FPType_SNaN || fpscr_val[26] == '1' then
14 FPProcessException(FPExc_InvalidOp, fpscr_val);
15 elsif fp_type == FPType_Infinity then
16 if fpscr_val[26] == '1' then // AH bit set
17 result = sign : Ones(15);
18 FPProcessException(FPExc_InvalidOp, fpscr_val);
19 else
20 result = FPInfinity(sign, 16);
21 elsif fp_type == FPType_Zero then
22 result = FPZero(sign, 16);
23 else
24 result = FPRound(value, 16, fpscr_val);
25 return result;

```

### E2.1.115 FPDoubleToSingle

```

1 // FPDoubleToSingle()
2 // =====
3
4 bits(32) FPDoubleToSingle(bits(64) operand, boolean fpscr_controlled)
5 fpscr_val = if fpscr_controlled then FPSCR else StandardFPSCRValue();
6 (fp_type,sign,value) = FPUnpack(operand, fpscr_val);
7 if fp_type == FPType_SNaN || fp_type == FPType_QNaN then
8 if fpscr_val[25] == '1' then // DN bit set
9 result = FPDefaultNaN(32);
10 else
11 result = sign : '11111111 1' : operand[50:29];
12 if fp_type == FPType_SNaN then
13 FPProcessException(FPExc_InvalidOp, fpscr_val);
14 elsif fp_type == FPType_Infinity then
15 result = FPInfinity(sign, 32);
16 elsif fp_type == FPType_Zero then
17 result = FPZero(sign, 32);
18 else
19 result = FPRound(value, 32, fpscr_val);
20 return result;

```

### E2.1.116 FPExc

```

1 // Floating point exceptions
2
3 enumeration FPExc {FPExc_InvalidOp, FPExc_DivideByZero, FPExc_Overflow,
4 FPExc_Underflow, FPExc_Inexact, FPExc_InputDenorm};

```

### E2.1.117 FPHalfToDouble

```

1 // FPHalfToDouble()
2 // =====
3
4 bits(64) FPHalfToDouble(bits(16) operand, boolean fpscr_controlled)
5 fpscr_val = if fpscr_controlled then FPSCR else StandardFPSCRValue();
6 (fp_type,sign,value) = FPUnpack(operand, fpscr_val);
7 if fp_type == FPType_SNaN || fp_type == FPType_QNaN then
8 if fpscr_val[25] == '1' then // DN bit set
9 result = FPDefaultNaN(64);
10 else
11 result = sign : '1111111111 1' : operand[8:0] : Zeros(42);

```

```

12 if fp_type == FPType_SNaN then
13 FPProcessException(FPExc_InvalidOp, fpscr_val);
14 elsif fp_type == FPType_Infinity then
15 result = FPInfinity(sign, 64);
16 elsif fp_type == FPType_Zero then
17 result = FPZero(sign, 64);
18 else
19 result = FPRound(value, 64, fpscr_val); // Rounding will be exact
20 return result;

```

### E2.1.118 FPHalfToSingle

```

1 // FPHalfToSingle()
2 // =====
3
4 bits(32) FPHalfToSingle(bits(16) operand, boolean fpscr_controlled)
5 fpscr_val = if fpscr_controlled then FPSCR else StandardFPSCRValue();
6 (fp_type,sign,value) = FPUnpack(operand, fpscr_val);
7 if fp_type == FPType_SNaN || fp_type == FPType_QNaN then
8 if fpscr_val[25] == '1' then // DN bit set
9 result = FPDefaultNaN(32);
10 else
11 result = sign : '11111111 1' : operand[8:0] : Zeros(13);
12 if fp_type == FPType_SNaN then
13 FPProcessException(FPExc_InvalidOp, fpscr_val);
14 elsif fp_type == FPType_Infinity then
15 result = FPInfinity(sign, 32);
16 elsif fp_type == FPType_Zero then
17 result = FPZero(sign, 32);
18 else
19 result = FPRound(value, 32, fpscr_val); // Rounding will be exact
20 return result;

```

### E2.1.119 FPInfinity

```

1 // FPInfinity()
2 // =====
3
4 bits(N) FPInfinity(bit sign, integer N)
5 assert N IN {16,32,64};
6 integer E = if N == 16 then 5 elsif N == 32 then 8 else 11;
7 constant integer F = N - E - 1;
8 bits(E) exp;
9 bits(F) frac;
10 exp = Ones(E);
11 frac = Zeros(F);
12 return sign : exp : frac;

```

### E2.1.120 FPMax

```

1 // FPMax()
2 // =====
3
4 bits(N) FPMax(bits(N) op1, bits(N) op2, boolean fpscr_controlled)
5 assert N IN {32,64};
6 fpscr_val = if fpscr_controlled then FPSCR else StandardFPSCRValue();
7 (fp_type1,sign1,value1) = FPUnpack(op1, fpscr_val);
8 (fp_type2,sign2,value2) = FPUnpack(op2, fpscr_val);
9 (done,result) = FPProcessNaNs(fp_type1, fp_type2, op1, op2, fpscr_val);
10 if !done then
11 if value1 > value2 then
12 (fp_type,sign,value) = (fp_type1,sign1,value1);
13 else
14 (fp_type,sign,value) = (fp_type2,sign2,value2);
15 if fp_type == FPType_Infinity then

```

```

16 result = FPInfinity(sign, N);
17 elseif fp_type == FPType_Zero then
18 sign = sign1 AND sign2; // Use most positive sign
19 result = FPZero(sign, N);
20 else
21 result = FPRound(value, N, fpscr_val);
22 return result;

```

### E2.1.121 FPMaxNormal

```

1 // FPMaxNormal()
2 // =====
3
4 bits(N) FPMaxNormal(bit sign, integer N)
5 assert N IN {16,32,64};
6 integer E = if N == 16 then 5 elseif N == 32 then 8 else 11;
7 constant integer F = N - E - 1;
8 bits(E) exp;
9 bits(F) frac;
10 exp = Ones(E-1):'0';
11 frac = Ones(F);
12 return sign : exp : frac;

```

### E2.1.122 FPMaxNum

```

1 // FPMaxNum()
2 // =====
3
4 bits(N) FPMaxNum(bits(N) op1, bits(N) op2)
5 assert N IN {32,64};
6
7 (type1,-,-) = FPUnpack(op1, FPSCR);
8 (type2,-,-) = FPUnpack(op2, FPSCR);
9
10 // treat a single quiet-NaN as -Infinity
11 if type1 == FPType_QNaN && type2 != FPType_QNaN then
12 op1 = FPInfinity('1', N);
13 elseif type1 != FPType_QNaN && type2 == FPType_QNaN then
14 op2 = FPInfinity('1', N);
15
16 return FPMax(op1, op2, TRUE);

```

### E2.1.123 FPMin

```

1 // FPMin()
2 // =====
3
4 bits(N) FPMin(bits(N) op1, bits(N) op2, boolean fpscr_controlled)
5 assert N IN {32,64};
6 fpscr_val = if fpscr_controlled then FPSCR else StandardFPSCRValue();
7 (fp_type1,sign1,value1) = FPUnpack(op1, fpscr_val);
8 (fp_type2,sign2,value2) = FPUnpack(op2, fpscr_val);
9 (done,result) = FPProcessNaNs(fp_type1, fp_type2, op1, op2, fpscr_val);
10 if !done then
11 if value1 < value2 then
12 (fp_type,sign,value) = (fp_type1,sign1,value1);
13 else
14 (fp_type,sign,value) = (fp_type2,sign2,value2);
15 if fp_type == FPType_Infinity then
16 result = FPInfinity(sign, N);
17 elseif fp_type == FPType_Zero then
18 sign = sign1 OR sign2; // Use most negative sign
19 result = FPZero(sign, N);
20 else
21 result = FPRound(value, N, fpscr_val);
22 return result;

```

**E2.1.124 FPMinNum**

```

1 // FPMinNum()
2 // =====
3
4 bits(N) FPMinNum(bits(N) op1, bits(N) op2)
5 assert N IN {32,64};
6
7 (fp_type1,-,-) = FPUnpack(op1, FPSCR);
8 (fp_type2,-,-) = FPUnpack(op2, FPSCR);
9
10 // Treat a single quiet-NaN as +Infinity
11 if fp_type1 == FPType_QNaN && fp_type2 != FPType_QNaN then
12 op1 = FPInfinity('0', N);
13 elsif fp_type1 != FPType_QNaN && fp_type2 == FPType_QNaN then
14 op2 = FPInfinity('0', N);
15
16 return FPMin(op1, op2, TRUE);

```

**E2.1.125 FPMul**

```

1 // FPMul()
2 // =====
3
4 bits(N) FPMul(bits(N) op1, bits(N) op2, boolean fpscr_controlled)
5 assert N IN {32,64};
6 fpscr_val = if fpscr_controlled then FPSCR else StandardFPSCRValue();
7 (type1,sign1,value1) = FPUnpack(op1, fpscr_val);
8 (type2,sign2,value2) = FPUnpack(op2, fpscr_val);
9 (done,result) = FPProcessNaNs(type1, type2, op1, op2, fpscr_val);
10 if !done then
11 inf1 = (type1 == FPType_Infinity); inf2 = (type2 == FPType_Infinity);
12 zero1 = (type1 == FPType_Zero); zero2 = (type2 == FPType_Zero);
13 if (inf1 && zero2) || (zero1 && inf2) then
14 result = FPDefaultNaN(N);
15 FPProcessException(FPExc_InvalidOp, fpscr_val);
16 elsif inf1 || inf2 then
17 result_sign = if sign1 == sign2 then '0' else '1';
18 result = FPInfinity(result_sign, N);
19 elsif zero1 || zero2 then
20 result_sign = if sign1 == sign2 then '0' else '1';
21 result = FPZero(result_sign, N);
22 else
23 result = FPRound(value1*value2, N, fpscr_val);
24 return result;

```

**E2.1.126 FPMulAdd**

```

1 // FPMulAdd()
2 // =====
3 // Calculates addend + op1*op2 with a single rounding.
4
5 bits(N) FPMulAdd(bits(N) addend, bits(N) op1, bits(N) op2,
6 boolean fpscr_controlled)
7 assert N IN {32,64};
8 fpscr_val = if fpscr_controlled then FPSCR else StandardFPSCRValue();
9 (typeA,signA,valueA) = FPUnpack(addend, fpscr_val);
10 (type1,sign1,value1) = FPUnpack(op1, fpscr_val);
11 (type2,sign2,value2) = FPUnpack(op2, fpscr_val);
12 inf1 = (type1 == FPType_Infinity); zero1 = (type1 == FPType_Zero);
13 inf2 = (type2 == FPType_Infinity); zero2 = (type2 == FPType_Zero);
14 (done,result) = FPProcessNaNs3(typeA, type1, type2, addend, op1, op2, fpscr_val);
15
16 if typeA == FPType_QNaN && ((inf1 && zero2) || (zero1 && inf2)) then
17 result = FPDefaultNaN(N);
18 FPProcessException(FPExc_InvalidOp, fpscr_val);

```



```

19
20 if !done then
21 infA = (typeA == FPType_Infinity); zeroA = (typeA == FPType_Zero);
22
23 // Determine sign and type product will have if it does not cause an Invalid
24 // Operation.
25 signP = if sign1 == sign2 then '0' else '1';
26 infP = inf1 || inf2;
27 zeroP = zero1 || zero2;
28
29 // Non SNaN-generated Invalid Operation cases are multiplies of zero by infinity and
30 // additions of opposite-signed infinities.
31 if (inf1 && zero2) || (zero1 && inf2) || (infA && infP && signA == NOT(signP)) then
32 result = FPDefaultNaN(N);
33 FPProcessException(FPExc_InvalidOp, fpscr_val);
34
35 // Other cases involving infinities produce an infinity of the same sign.
36 elsif (infA && signA == '0') || (infP && signP == '0') then
37 result = FPInfinity('0', N);
38 elsif (infA && signA == '1') || (infP && signP == '1') then
39 result = FPInfinity('1', N);
40
41 // Cases where the result is exactly zero and its sign is not determined by the
42 // rounding mode are additions of same-signed zeros.
43 elsif zeroA && zeroP && signA == signP then
44 result = FPZero(signA, N);
45
46 // Otherwise calculate numerical result and round it.
47 else
48 result_value = valueA + (value1 * value2);
49 if result_value == 0.0 then // Sign of exact zero result depends on rounding
 mode
50 result_sign = if fpscr_val[23:22] == '10' then '1' else '0';
51 result = FPZero(result_sign, N);
52 else
53 result = FPRound(result_value, N, fpscr_val);
54
55 return result;

```

### E2.1.127 FPNeg

```

1 // FPNeg()
2 // =====
3
4 bits(N) FPNeg(bits(N) operand)
5 assert N IN {32,64};
6 return NOT(operand[N-1]) : operand[N-2:0];

```

### E2.1.128 FPProcessException

```

1 // FPProcessException()
2 // =====
3 // The 'fpscr_val' argument supplies FPSCR control bits. Status information is
4 // updated directly in FPSCR where appropriate.
5
6 FPProcessException(FPExc exception, bits(32) fpscr_val)
7 // Get appropriate FPSCR bit numbers
8 case exception of
9 when FPExc_InvalidOp enable = 8; cumul = 0;
10 when FPExc_DivideByZero enable = 9; cumul = 1;
11 when FPExc_Overflow enable = 10; cumul = 2;
12 when FPExc_Underflow enable = 11; cumul = 3;
13 when FPExc_Inexact enable = 12; cumul = 4;
14 when FPExc_InputDenorm enable = 15; cumul = 7;
15 if fpscr_val[enable] == '1' then
16 IMPLEMENTATION_DEFINED "floating-point trap handling";
17 else

```

```

18 FPSCR[cumul] = '1';
19 return;

```

### E2.1.129 FPProcessNaN

```

1 // FPProcessNaN()
2 // =====
3 // The 'fpscr_val' argument supplies FPSCR control bits. Status information is
4 // updated directly in FPSCR where appropriate.
5
6 bits(N) FPProcessNaN(FPType fp_type, bits(N) operand, bits(32) fpscr_val)
7 assert N IN {32,64};
8 topfrac = if N == 32 then 22 else 51;
9 result = operand;
10 if fp_type == FPType_SNaN then
11 result[topfrac] = '1';
12 FPProcessException(FPExc_InvalidOp, fpscr_val);
13 if fpscr_val[25] == '1' then // DefaultNaN requested
14 result = FPDefaultNaN(N);
15 return result;

```

### E2.1.130 FPProcessNaNs

```

1 // FPProcessNaNs()
2 // =====
3 // The boolean part of the return value says whether a NaN has been found and
4 // processed. The bits(N) part is only relevant if it has and supplies the
5 // result of the operation.
6 //
7 // The 'fpscr_val' argument supplies FPSCR control bits. Status information is
8 // updated directly in FPSCR where appropriate.
9
10 (boolean, bits(N)) FPProcessNaNs(FPType type1, FPType type2,
11 bits(N) op1, bits(N) op2,
12 bits(32) fpscr_val)
13 assert N IN {32,64};
14 if type1 == FPType_SNaN then
15 done = TRUE; result = FPProcessNaN(type1, op1, fpscr_val);
16 elseif type2 == FPType_SNaN then
17 done = TRUE; result = FPProcessNaN(type2, op2, fpscr_val);
18 elseif type1 == FPType_QNaN then
19 done = TRUE; result = FPProcessNaN(type1, op1, fpscr_val);
20 elseif type2 == FPType_QNaN then
21 done = TRUE; result = FPProcessNaN(type2, op2, fpscr_val);
22 else
23 done = FALSE; result = Zeros(N); // 'Don't care' result
24 return (done, result);

```

### E2.1.131 FPProcessNaNs3

```

1 // FPProcessNaNs3()
2 // =====
3 // The boolean part of the return value says whether a NaN has been found and
4 // processed. The bits(N) part is only relevant if it has and supplies the
5 // result of the operation.
6 //
7 // The 'fpscr_val' argument supplies FPSCR control bits. Status information is
8 // updated directly in FPSCR where appropriate.
9
10 (boolean, bits(N)) FPProcessNaNs3(FPType type1, FPType type2, FPType type3,
11 bits(N) op1, bits(N) op2, bits(N) op3,
12 bits(32) fpscr_val)
13 assert N IN {32,64};
14 if type1 == FPType_SNaN then
15 done = TRUE; result = FPProcessNaN(type1, op1, fpscr_val);

```

```

16 elsif type2 == FPType_SNaN then
17 done = TRUE; result = FPProcessNaN(type2, op2, fpscr_val);
18 elsif type3 == FPType_SNaN then
19 done = TRUE; result = FPProcessNaN(type3, op3, fpscr_val);
20 elsif type1 == FPType_QNaN then
21 done = TRUE; result = FPProcessNaN(type1, op1, fpscr_val);
22 elsif type2 == FPType_QNaN then
23 done = TRUE; result = FPProcessNaN(type2, op2, fpscr_val);
24 elsif type3 == FPType_QNaN then
25 done = TRUE; result = FPProcessNaN(type3, op3, fpscr_val);
26 else
27 done = FALSE; result = Zeros(N); // 'Don't care' result
28 return (done, result);

```

### E2.1.132 FPRound

```

1 // FPRound()
2 // =====
3 // The 'fpscr_val' argument supplies FPSCR control bits. Status information is
4 // updated directly in FPSCR where appropriate.
5
6 bits(N) FPRound(real value, integer N, bits(32) fpscr_val)
7 assert N IN {16,32,64};
8 assert value != 0.0;
9
10 // Obtain format parameters - minimum exponent, numbers of exponent and fraction bits.
11 integer E = if N == 16 then 5 elsif N == 32 then 8 else 11;
12 minimum_exp = 2 - 2^(E-1);
13 constant integer F = N - E - 1;
14
15 // Split value into sign, unrounded mantissa and exponent.
16 if value < 0.0 then
17 sign = '1'; mantissa = -value;
18 else
19 sign = '0'; mantissa = value;
20 exponent = 0;
21 while mantissa < 1.0 do
22 mantissa = mantissa * 2.0; exponent = exponent - 1;
23 while mantissa >= 2.0 do
24 mantissa = mantissa / 2.0; exponent = exponent + 1;
25
26 // Deal with flush-to-zero.
27 if fpscr_val[24] == '1' && N != 16 && exponent < minimum_exp then
28 result = FPZero(sign, N);
29 FPSCR.UFC = '1'; // Flush-to-zero never generates a trapped exception
30 else
31
32 // Start creating the exponent value for the result. Start by biasing the actual
33 // so that the minimum exponent becomes 1, lower values 0 (indicating possible
34 // underflow).
35 biased_exp = Max(exponent - minimum_exp + 1, 0);
36 if biased_exp == 0 then mantissa = mantissa / 2.0^(minimum_exp - exponent);
37
38 // Get the unrounded mantissa as an integer, and the "units in last place" rounding
39 // error.
40 int_mant = RoundDown(mantissa * 2.0^F); // < 2.0^F if biased_exp == 0, >= 2.0^F if
41 // not
42 error = mantissa * 2.0^F - Real(int_mant);
43
44 // Underflow occurs if exponent is too small before rounding, and result is inexact
45 // or
46 // the Underflow exception is trapped.
47 if biased_exp == 0 && (error != 0.0 || fpscr_val[11] == '1') then
48 FPProcessException(FPExc_Underflow, fpscr_val);
49
50 // Round result according to rounding mode.
51 case fpscr_val[23:22] of

```

```

48 when '00' // Round to Nearest (rounding to even if exactly halfway)
49 round_up = (error > 0.5 || (error == 0.5 && int_mant[0] == '1'));
50 overflow_to_inf = TRUE;
51 when '01' // Round towards Plus Infinity
52 round_up = (error != 0.0 && sign == '0');
53 overflow_to_inf = (sign == '0');
54 when '10' // Round towards Minus Infinity
55 round_up = (error != 0.0 && sign == '1');
56 overflow_to_inf = (sign == '1');
57 when '11' // Round towards Zero
58 round_up = FALSE;
59 overflow_to_inf = FALSE;
60 if round_up then
61 int_mant = int_mant + 1;
62 if int_mant == 2^F then // Rounded up from denormalized to normalized
63 biased_exp = 1;
64 if int_mant == 2^(F+1) then // Rounded up to next exponent
65 biased_exp = biased_exp + 1; int_mant = int_mant DIV 2;
66
67 // Deal with overflow and generate result.
68 if N != 16 || fpscr_val[26] == '0' then // Single, double or IEEE half precision
69 if biased_exp >= 2^E - 1 then
70 result = if overflow_to_inf then FPInfinity(sign, N) else FPMaxNormal(sign, N
71);
72 FPProcessException(FPExc_Overflow, fpscr_val);
73 error = 1.0; // Ensure that an Inexact exception occurs
74 else
75 result = sign : biased_exp[E-1:0] : int_mant[F-1:0];
76 else // Alternative half precision
77 if biased_exp >= 2^E then
78 result = sign : Ones(N-1);
79 FPProcessException(FPExc_InvalidOp, fpscr_val);
80 error = 0.0; // Ensure that an Inexact exception does not occur
81 else
82 result = sign : biased_exp[E-1:0] : int_mant[F-1:0];
83
84 // Deal with Inexact exception.
85 if error != 0.0 then
86 FPProcessException(FPExc_Inexact, fpscr_val);
87
88 return result;

```

### E2.1.133 FPRoundInt

```

1 // FPRoundInt()
2 // =====
3 // Round floating-point value to nearest integral floating point value
4 // using given rounding mode. If exact is TRUE, set inexact flag if result
5 // is not numerically equal to given value.
6
7 bits(N) FPRoundInt(bits(N) op, bits(2) rmode, boolean away, boolean exact)
8 assert N IN {32,64};
9
10 // Unpack using FPSCR to determine if subnormals are flushed-to-zero
11 (fp_type, sign, value) = FPUnpack(op, FPSCR);
12
13 if fp_type == FPType_SNaN || fp_type == FPType_QNaN then
14 result = FPProcessNaN(fp_type, op, FPSCR);
15 elseif fp_type == FPType_Infinity then
16 result = FPInfinity(sign, N);
17 elseif fp_type == FPType_Zero then
18 result = FPZero(sign, N);
19 else
20 // extract integer component
21 int_result = RoundDown(value);
22 error = value - Real(int_result);
23
24 // Determine whether supplied rounding mode requires an increment

```

```

25 case rmode of
26 when '00' // Round to nearest, ties to even
27 round_up = (error > 0.5 || (error == 0.5 && int_result[0] == '1'));
28 when '01' // Round towards Plus Infinity
29 round_up = (error != 0.0);
30 when '10' // Round towards Minus Infinity
31 round_up = FALSE;
32 when '11' // Round towards Zero
33 round_up = (error != 0.0 && int_result < 0);
34
35 if away then // Round towards Zero, ties away
36 round_up = (error > 0.5 || (error == 0.5 && int_result >= 0));
37
38 if round_up then int_result = int_result + 1;
39
40 // Convert integer value into an equivalent real value
41 real_result = Real(int_result);
42
43 // Re-encode as a floating-point value, result is always exact
44 if real_result == 0.0 then
45 result = FPZero(sign, N);
46 else
47 result = FPRound(real_result, N, FPSCR);
48
49 // Generate inexact exceptions
50 if error != 0.0 && exact then
51 FPProcessException(FPExc_Inexact, FPSCR);
52
53 return result;

```

### E2.1.134 FPSingleToDouble

```

1 // FPSingleToDouble()
2 // =====
3
4 bits(64) FPSingleToDouble(bits(32) operand, boolean fpscr_controlled)
5 fpscr_val = if fpscr_controlled then FPSCR else StandardFPSCRValue();
6 (fp_type, sign, value) = FPUnpack(operand, fpscr_val);
7 if fp_type == FPType_SNaN || fp_type == FPType_QNaN then
8 if fpscr_val[25] == '1' then // DN bit set
9 result = FPDefaultNaN(64);
10 else
11 result = sign : '1111111111 1' : operand[21:0] : Zeros(29);
12 if fp_type == FPType_SNaN then
13 FPProcessException(FPExc_InvalidOp, fpscr_val);
14 elsif fp_type == FPType_Infinity then
15 result = FPInfinity(sign, 64);
16 elsif fp_type == FPType_Zero then
17 result = FPZero(sign, 64);
18 else
19 result = FPRound(value, 64, fpscr_val); // Rounding will be exact
20 return result;

```

### E2.1.135 FPSingleToHalf

```

1 // FPSingleToHalf()
2 // =====
3
4 bits(16) FPSingleToHalf(bits(32) operand, boolean fpscr_controlled)
5 fpscr_val = if fpscr_controlled then FPSCR else StandardFPSCRValue();
6 (fp_type, sign, value) = FPUnpack(operand, fpscr_val);
7 if fp_type == FPType_SNaN || fp_type == FPType_QNaN then
8 if fpscr_val[26] == '1' then // AH bit set
9 result = FPZero(sign, 16);
10 elsif fpscr_val[25] == '1' then // DN bit set
11 result = FPDefaultNaN(16);
12 else

```

```

13 result = sign : '11111 1' : operand[21:13];
14 if fp_type == FPType_SNaN || fp_scr_val[26] == '1' then
15 FPProcessException(FPExc_InvalidOp, fp_scr_val);
16 elsif fp_type == FPType_Infinity then
17 if fp_scr_val[26] == '1' then // AH bit set
18 result = sign : Ones(15);
19 FPProcessException(FPExc_InvalidOp, fp_scr_val);
20 else
21 result = FPInfinity(sign, 16);
22 elsif fp_type == FPType_Zero then
23 result = FPZero(sign, 16);
24 else
25 result = FPRound(value, 16, fp_scr_val);
26 return result;

```

### E2.1.136 FPSqrt

```

1 // FPSqrt()
2 // =====
3
4 bits(N) FPSqrt(bits(N) operand)
5 assert N IN {32,64};
6 (fp_type,sign,value) = FPUnpack(operand, FPSCR);
7 if fp_type == FPType_SNaN || fp_type == FPType_QNaN then
8 result = FPProcessNaN(fp_type, operand, FPSCR);
9 elsif fp_type == FPType_Zero then
10 result = FPZero(sign, N);
11 elsif fp_type == FPType_Infinity && sign == '0' then
12 result = FPInfinity(sign, N);
13 elsif sign == '1' then
14 result = FPDefaultNaN(N);
15 FPProcessException(FPExc_InvalidOp, FPSCR);
16 else
17 result = FPRound(Sqrt(value), N, FPSCR);
18 return result;

```

### E2.1.137 FPSub

```

1 // FPSub()
2 // =====
3
4 bits(N) FPSub(bits(N) op1, bits(N) op2, boolean fp_scr_controlled)
5 assert N IN {32,64};
6 fp_scr_val = if fp_scr_controlled then FPSCR else StandardFPSCRValue();
7 (fp_type1,sign1,value1) = FPUnpack(op1, fp_scr_val);
8 (fp_type2,sign2,value2) = FPUnpack(op2, fp_scr_val);
9 (done,result) = FPProcessNaNs(fp_type1, fp_type2, op1, op2, fp_scr_val);
10 if !done then
11 inf1 = (fp_type1 == FPType_Infinity); inf2 = (fp_type2 == FPType_Infinity);
12 zero1 = (fp_type1 == FPType_Zero); zero2 = (fp_type2 == FPType_Zero);
13 if inf1 && inf2 && sign1 == sign2 then
14 result = FPDefaultNaN(N);
15 FPProcessException(FPExc_InvalidOp, fp_scr_val);
16 elsif (inf1 && sign1 == '0') || (inf2 && sign2 == '1') then
17 result = FPInfinity('0', N);
18 elsif (inf1 && sign1 == '1') || (inf2 && sign2 == '0') then
19 result = FPInfinity('1', N);
20 elsif zero1 && zero2 && sign1 == NOT(sign2) then
21 result = FPZero(sign1, N);
22 else
23 result_value = value1 - value2;
24 if result_value == 0.0 then // Sign of exact zero result depends on rounding
25 mode
26 result_sign = if fp_scr_val[23:22] == '10' then '1' else '0';
27 result = FPZero(result_sign, N);
28 else
29 result = FPRound(result_value, N, fp_scr_val);

```

```
29 return result;
```

### E2.1.138 FPToFixed

```
1 // FPToFixed()
2 // =====
3
4 bits(M) FPToFixed(bits(N) operand, integer M, integer fraction_bits, boolean unsigned,
5 boolean round_towards_zero, boolean fpscr_controlled)
6 assert N IN {32,64};
7 fpscr_val = if fpscr_controlled then FPSCR else StandardFPSCRValue();
8 if round_towards_zero then fpscr_val[23:22] = '11';
9 (fp_type,-,value) = FPUnpack(operand, fpscr_val);
10
11 // For NaNs and infinities, FPUnpack() has produced a value that will round to the
12 // required result of the conversion. Also, the value produced for infinities will
13 // cause the conversion to overflow and signal an Invalid Operation floating-point
14 // exception as required. NaNs must also generate such a floating-point exception.
15 if fp_type == FPType_SNaN || fp_type == FPType_QNaN then
16 FPProcessException(FPExc_InvalidOp, fpscr_val);
17
18 // Scale value by specified number of fraction bits, then start rounding to an integer
19 // and determine the rounding error.
20 value = value * 2.0^fraction_bits;
21 int_result = RoundDown(value);
22 error = value - Real(int_result);
23
24 // Apply the specified rounding mode.
25 case fpscr_val[23:22] of
26 when '00' // Round to Nearest (rounding to even if exactly halfway)
27 round_up = (error > 0.5 || (error == 0.5 && int_result[0] == '1'));
28 when '01' // Round towards Plus Infinity
29 round_up = (error != 0.0);
30 when '10' // Round towards Minus Infinity
31 round_up = FALSE;
32 when '11' // Round towards Zero
33 round_up = (error != 0.0 && int_result < 0);
34 if round_up then int_result = int_result + 1;
35
36 // Bitstring result is the integer result saturated to the destination size, with
37 // saturation indicating overflow of the conversion (signaled as an Invalid
38 // Operation floating-point exception).
39 (result, overflow) = SatQ(int_result, M, unsigned);
40 if overflow then
41 FPProcessException(FPExc_InvalidOp, fpscr_val);
42 elseif error != 0.0 then
43 FPProcessException(FPExc_Inexact, fpscr_val);
44
45 return result;
```

### E2.1.139 FPToFixedDirected

```
1 // FPToFixedDirected()
2 // =====
3
4 bits(M) FPToFixedDirected(bits(N) op, integer fbits, boolean unsigned,
5 bits(2) round_mode, boolean fpscr_controlled)
6 assert N IN {32,64};
7
8 fpscr_val = if fpscr_controlled then FPSCR else StandardFPSCRValue();
9
10 // Unpack using FPSCR to determine if subnormals are flushed-to-zero
11 (fp_type,-,value) = FPUnpack(op, fpscr_val);
12
13 // If NaN, set cumulative flag or take exception
14 if fp_type == FPType_SNaN || fp_type == FPType_QNaN then
15 FPProcessException(FPExc_InvalidOp, FPSCR);
```

```

16
17 // Scale by fractional bits and produce integer rounded towards
18 // minus-infinity
19 value = value * 2.0^fbits;
20 int_result = RoundDown(value);
21 error = value - Real(int_result);
22
23 // Determine whether supplied rounding mode requires an increment
24 case round_mode of
25 when '00' // ties away
26 round_up = (error > 0.5 || (error == 0.5 && int_result >= 0));
27 when '01' // nearest even
28 round_up = (error > 0.5 || (error == 0.5 && int_result[0] == '1'));
29 when '10' // plus infinity
30 round_up = (error != 0.0);
31 when '11' // neg infinity
32 round_up = FALSE;
33
34 if round_up then int_result = int_result + 1;
35
36 // Generate saturated result and exceptions
37 (result, overflow) = SatQ(int_result, M, unsigned);
38
39 if overflow then
40 FPProcessException(FPExc_InvalidOp, fpscr_val);
41 elseif error != 0.0 then
42 FPProcessException(FPExc_Inexact, fpscr_val);
43 return result;

```

### E2.1.140 FPType

```

1 // Type of floating-point value. Floating-point values are categorized into one
2 // of the following type during unpacking.
3
4 enumeration FPType {FPType_Nonzero, FPType_Zero, FPType_Infinity, FPType_QNaN, FPType_SNaN};

```

### E2.1.141 FPUnpack

```

1 // FPUnpack()
2 // =====
3 //
4 // Unpack a floating-point number into its type, sign bit and the real number
5 // that it represents. The real number result has the correct sign for numbers
6 // and infinities, is very large in magnitude for infinities, and is 0.0 for
7 // NaNs. (These values are chosen to simplify the description of comparisons
8 // and conversions.)
9 //
10 // The 'fpscr_val' argument supplies FPSCR control bits. Status information is
11 // updated directly in FPSCR where appropriate.
12
13 (FPType, bit, real) FPUnpack(bits(N) fpval, bits(32) fpscr_val)
14 assert N IN {16,32,64};
15
16 if N == 16 then
17 sign = fpval[15];
18 exp16 = fpval[14:10];
19 frac16 = fpval[9:0];
20 if IsZero(exp16) then
21 // Produce zero if value is zero
22 if IsZero(frac16) then
23 fp_type = FPType_Zero; value = 0.0;
24 else
25 fp_type = FPType_Nonzero; value = 2.0^-14 * (Real(UInt(frac16)) * 2.0^-10);
26 elseif IsOnes(exp16) && fpscr_val[26] == '0' then // Infinity or NaN in IEEE format
27 if IsZero(frac16) then
28 fp_type = FPType_Infinity; value = 2.0^1000000;
29 else

```



```

30 fp_type = if frac16[9] == '1' then FPType_QNaN else FPType_SNaN;
31 value = 0.0;
32 else
33 fp_type = FPType_Nonzero;
34 value = 2.0^(UInt(exp16)-15) * (1.0 + Real(UInt(frac16)) * 2.0^-10);
35
36 elsif N == 32 then
37
38 sign = fpval[31];
39 exp32 = fpval[30:23];
40 frac32 = fpval[22:0];
41 if IsZero(exp32) then
42 // Produce zero if value is zero or flush-to-zero is selected.
43 if IsZero(frac32) || fpscr_val[24] == '1' then
44 fp_type = FPType_Zero; value = 0.0;
45 if !IsZero(frac32) then // Denormalized input flushed to zero
46 FPProcessException(FPExc_InputDenorm, fpscr_val);
47 else
48 fp_type = FPType_Nonzero; value = 2.0^-126 * (Real(UInt(frac32)) * 2.0^-23);
49 elsif IsOnes(exp32) then
50 if IsZero(frac32) then
51 fp_type = FPType_Infinity; value = 2.0^1000000;
52 else
53 fp_type = if frac32[22] == '1' then FPType_QNaN else FPType_SNaN;
54 value = 0.0;
55 else
56 fp_type = FPType_Nonzero;
57 value = 2.0^(UInt(exp32)-127) * (1.0 + Real(UInt(frac32)) * 2.0^-23);
58
59 else // N == 64
60
61 sign = fpval[63];
62 exp64 = fpval[62:52];
63 frac64 = fpval[51:0];
64 if IsZero(exp64) then
65 // Produce zero if value is zero or flush-to-zero is selected.
66 if IsZero(frac64) || fpscr_val[24] == '1' then
67 fp_type = FPType_Zero; value = 0.0;
68 if !IsZero(frac64) then // Denormalized input flushed to zero
69 FPProcessException(FPExc_InputDenorm, fpscr_val);
70 else
71 fp_type = FPType_Nonzero; value = 2.0^-1022 * (Real(UInt(frac64)) * 2.0^-52)
72 ;
73 elsif IsOnes(exp64) then
74 if IsZero(frac64) then
75 fp_type = FPType_Infinity; value = 2.0^1000000;
76 else
77 fp_type = if frac64[51] == '1' then FPType_QNaN else FPType_SNaN;
78 value = 0.0;
79 else
80 fp_type = FPType_Nonzero;
81 value = 2.0^(UInt(exp64)-1023) * (1.0 + Real(UInt(frac64)) * 2.0^-52);
82
83 if sign == '1' then value = -value;
84 return (fp_type, sign, value);

```

### E2.1.142 FPZero

```

1 // FPZero()
2 // =====
3
4 bits(N) FPZero(bit sign, integer N)
5 assert N IN {16,32,64};
6 integer E = if N == 16 then 5 elsif N == 32 then 8 else 11;
7 constant integer F = N - E - 1;
8 bits(E) exp;
9 bits(F) frac;
10 exp = Zeros(E);

```

```

11 frac = Zeros(F);
12 return sign : exp : frac;

```

### E2.1.143 FunctionReturn

```

1 // FunctionReturn()
2 // =====
3
4 ExcInfo FunctionReturn()
5 exc = DefaultExcInfo();
6
7 // Pull the return address and IPSR off the Secure stack
8 mode = CurrentMode();
9 spName = LookUpSP_with_security_mode(TRUE, mode);
10 framePtr = _SP(spName);
11 if !IsAligned(framePtr, 8) then UNPREDICTABLE;
12 // Only stack locations, not the load order are architected
13 RETPSR_Type newPSR;
14 if exc.fault == NoFault then (exc, newPSR) = Stack(framePtr, 4, spName, mode);
15 if exc.fault == NoFault then (exc, newPC) = Stack(framePtr, 0, spName, mode);
16
17 // Check the IPSR value that has been unstacked is consistent with the current
18 // mode, and being originally called from the Secure state.
19 // NOTE: It is IMPLEMENTATION DEFINED whether this check is performed before
20 // or after the load of the return address above.
21 if (exc.fault == NoFault) &&
22 !(((IPSR.Exception == 0[8:0]) && (newPSR.Exception == 0[8:0])) ||
23 ((IPSR.Exception == 1[8:0]) && (newPSR.Exception != 0[8:0]))) then
24 if HaveMainExt() then
25 UFSR_S.INVPC = '1';
26 // Create the exception. NOTE: If Main Extension not implemented then the fault
27 // always escalates to a HardFault
28 exc = CreateException(UsageFault, TRUE, TRUE);
29 // The IPSR value is set as UNKNOWN if the IPSR value is not supported by the PE
30 excNum = UInt(newPSR.Exception);
31 validIPSR = excNum IN {0, 1, NMI, HardFault, SVCcall, PendSV, SysTick};
32 if !validIPSR && HaveMainExt() then
33 validIPSR = excNum IN {MemManage, BusFault, UsageFault, SecureFault, DebugMonitor};
34 if !validIPSR && !IsIrqValid(excNum) then
35 newPSR.Exception = bits(9) UNKNOWN;
36
37 // Only consume the function return stack frame and update the XPSR/PC if no
38 // faults occurred.
39 if exc.fault == NoFault then
40 // Transition to the Secure state
41 CurrentState = SecurityState_Secure;
42 // Update stack pointer. NOTE: Stack pointer limit not checked on function
43 // return as stack pointer guaranteed to be ascending not descending.
44 _R[spName] = framePtr + 8;
45 IPSR.Exception = newPSR.Exception;
46 CONTROL_S.SFPA = newPSR.SFPA;
47 // IT/ICI bits cleared to prevent Non-secure code interfering with
48 // Secure execution
49 if HaveMainExt() then
50 ITSTATE = Zeros(8);
51 // if EPSR.T == 0, a UsageFault('Invalid State') or a HardFault is taken
52 // on the next instruction depending on whether the Main Extension is
53 // is implemented or not.
54 EPSR.T = newPC[0];
55 BranchTo(newPC[31:1]:'0');
56 return exc;

```

### E2.1.144 GenerateCoproprocessorException

```

1 // GenerateCoproprocessorException()
2 // =====
3

```

```

4 GenerateCoproprocessorException()
5 UFSR.UNDEFINSTR = '1';
6 excInfo = CreateException(UsageFault, FALSE, boolean UNKNOWN);
7 HandleException(excInfo);

```

### E2.1.145 GenerateDebugEventResponse

```

1 // GenerateDebugEventResponse()
2 // =====
3 // Generate a debug event response based on the PE configuration.
4
5 boolean GenerateDebugEventResponse()
6 if CanHaltOnEvent(IsSecure()) then
7 DFSR.BKPT = '1';
8 DHCSR.C_HALT = '1';
9 return TRUE;
10 elseif HaveMainExt() && CanPendMonitorOnEvent(IsSecure(), TRUE) then
11 DFSR.BKPT = '1';
12 DEMCR.MON_PEND = '1';
13 excInfo = CreateException(DebugMonitor, FALSE, boolean UNKNOWN);
14 HandleException(excInfo);
15 return TRUE;
16 else
17 return FALSE;

```

### E2.1.146 GenerateIntegerZeroDivide

```

1 // GenerateIntegerZeroDivide()
2 // =====
3
4 GenerateIntegerZeroDivide()
5 UFSR.DIVBYZERO = '1';
6 excInfo = CreateException(UsageFault, FALSE, boolean UNKNOWN);
7 HandleException(excInfo);

```

### E2.1.147 HaltingDebugAllowed

```

1 // HaltingDebugAllowed()
2 // =====
3
4 boolean HaltingDebugAllowed()
5 return ExternalInvasiveDebugEnabled() || DHCSR.S_HALT == '1';

```

### E2.1.148 HandleException

```

1 // HandleException()
2 // =====
3
4 HandleException(ExcInfo excInfo)
5 if excInfo.fault != NoFault then
6 if excInfo.lockup then
7 Lockup(excInfo.termInst);
8 else
9 // If the fault escalated to a HardFault update the syndrome info
10 if HaveMainExt() &&
11 (excInfo.fault == HardFault) &&
12 (excInfo.origFault != HardFault) then
13 HFSR.FORCED = '1';
14 // If the exception does not cause a lockup set the exception pending
15 // and potentially terminate execution of the current instruction
16 SetPending(excInfo.fault, excInfo.isSecure, TRUE);
17 if excInfo.termInst then
18 EndOfInstruction();

```

**E2.1.149 HaveDebugMonitor**

```

1 // HaveDebugMonitor()
2 //=====
3
4 boolean HaveDebugMonitor()
5 return HaveMainExt();

```

**E2.1.150 HaveDSPExt**

```

1 // HaveDSPExt()
2 // =====
3 // Check whether DSP Extension is implemented.
4
5 boolean HaveDSPExt();

```

**E2.1.151 HaveDWT**

```

1 // HaveDWT()
2 // =====
3 // Check whether Data Watchpoint and Trace unit is implemented.
4
5 boolean HaveDWT();

```

**E2.1.152 HaveFPB**

```

1 // HaveFPB()
2 // =====
3 // Check whether Flash Patch and Breakpoint unit is implemented.
4
5 boolean HaveFPB();

```

**E2.1.153 HaveFPExt**

```

1 // HaveFPExt()
2 // =====
3 // Check whether Floating Point Extension is implemented.
4
5 boolean HaveFPExt();

```

**E2.1.154 HaveHaltingDebug**

```

1 // HaveHaltingDebug()
2 // =====
3 // Check whether Halting debug implemented.
4
5 boolean HaveHaltingDebug();

```

**E2.1.155 HaveITM**

```

1 // HaveITM()
2 // =====
3 // Check whether Instrumentation Trace Macrocell is implemented.
4
5 boolean HaveITM();

```

**E2.1.156 HaveMainExt**

```

1 // HaveMainExt()
2 // =====
3 // Check whether Main Extension is implemented.
4
5 boolean HaveMainExt();

```

**E2.1.157 HaveSecurityExt**

```

1 // HaveSecurityExt()
2 // =====
3 // Check whether the implementation have Security Extensions.
4
5 boolean HaveSecurityExt();

```

**E2.1.158 HaveSPFPOnly**

```

1 // HaveSPFPOnly()
2 // =====
3 // Check whether Floating Point Extension only implements single-precision.
4
5 boolean HaveSPFPOnly();

```

**E2.1.159 HaveSysTick**

```

1 // HaveSysTick()
2 // =====
3 // Returns the number of SysTick instances (0, 1 or 2).
4
5 integer HaveSysTick();

```

**E2.1.160 HighestPri**

```

1 // HighestPri()
2 // =====
3 // Priority of Thread mode with no active exceptions.
4
5 integer HighestPri()
6 // The value is PriorityMax + 1 = 256 (configurable priority maximum bit field is 8 bits)
7 return 256;

```

**E2.1.161 HighestSetBit**

```

1 // HighestSetBit()
2 // =====
3
4 integer HighestSetBit(bits(N) x)
5 for i = N-1 downto 0
6 if x[i] == '1' then return i;
7 return -1;

```

**E2.1.162 Hint\_Debug**

```

1 // Hint_Debug
2 // =====
3 // Generate a hint to the debug system.
4
5 Hint_Debug(bits(4) option);

```

**E2.1.163 Hint\_PreloadData**

```

1 // Hint_PreloadData
2 // =====
3 // Performs a preload data hint
4
5 Hint_PreloadData(bits(32) address);

```

### E2.1.164 Hint\_PreloadDataForWrite

```

1 // Hint_PreloadDataForWrite
2 // =====
3 // Performs a preload data hint for write.
4
5 Hint_PreloadDataForWrite(bits(32) address);

```

### E2.1.165 Hint\_PreloadInstr

```

1 // Hint_PreloadInstr
2 // =====
3 // Performs a preload instructions hint
4
5 Hint_PreloadInstr(bits(32) address);

```

### E2.1.166 Hint\_Yield

```

1 // Hint_Yield
2 // =====
3 // Performs a Yield hint
4
5 Hint_Yield();

```

### E2.1.167 IDAUCheck

```

1 // IDAUCheck
2 // =====
3 // Query IDAU(Implementation Defined Attribution Unit) for attribution information
4
5 (boolean, boolean, boolean, bits(8), boolean) IDAUCheck(bits(32) address);

```

### E2.1.168 InITBlock

```

1 // InITBlock()
2 // =====
3
4 boolean InITBlock()
5 return (ITSTATE[3:0] != '0000');

```

### E2.1.169 InstructionAdvance

```

1 // InstructionAdvance()
2 // =====
3
4 InstructionAdvance(boolean instExecOk)
5 // Check for, and process any exception returns that were requested. This
6 // must be done after the instruction has completed so any exceptions
7 // raised during the exception return do not interfere with the execution of
8 // the instruction that cause the exception return (eg a POP causing an
9 // excReturn value to be written to the PC must adjust SP even if the
10 // exception return caused by the POP raises a fault).
11 excRetFault = FALSE;
12 EXC_RETURN_Type excReturn = NextInstrAddr();
13 if _PendingReturnOperation then

```

```

14 _PendingReturnOperation = FALSE;
15 (excInfo, excReturn) = ExceptionReturn(excReturn);
16 // Handle any faults raised during exception return
17 if excInfo.fault != NoFault then
18 excRetFault = TRUE;
19 // Either lockup, or pend the fault if it can be taken
20 if excInfo.lockup then
21 // Check if the fault occurred on exception return, or whether it
22 // occurred during a tail chained exception entry. This is
23 // because Lockups on exception return have to be handled
24 // differently.
25 if !excInfo.inExcTaken then
26 // If the fault occurred during exception return then the
27 // register state is UNKNOWN. This is due to the fact that
28 // an unknown amount of the exception stack frame might have
29 // been restored.
30 for n = 0 to 12
31 R[n] = bits(32) UNKNOWN;
32 LR = bits(32) UNKNOWN;
33 XPSR = bits(32) UNKNOWN;
34 if HaveFPEExt() then
35 for n = 0 to 31
36 S[n] = bits(32) UNKNOWN;
37 FPSCR = bits(32) UNKNOWN;
38 // If lockup is entered as a result of an exception return
39 // fault the original exception is deactivated. Therefore
40 // the stack pointer must be updated to consume the
41 // exception stack frame to keep the stack depth consistent
42 // with the number of active exceptions. NOTE: The XPSR SP
43 // alignment flag is UNKNOWN, assume it was zero.
44 ConsumeExcStackFrame(excReturn, '0');
45 // IPSR from stack is UNKNOWN, set IPSR based on mode
46 // specified in EXC_RETURN.
47 IPSR.Exception = (if excReturn.Mode == '1' then NoFault else HardFault)
48 [8:0];
49 if HaveFPEExt() then
50 CONTROL.FPCA = NOT(excReturn.FType);
51 CONTROL_S.SFPA = bit UNKNOWN;
52 Lockup(FALSE);
53 else
54 // Set syndrome if fault escalated to a HardFault
55 if HaveMainExt() &&
56 (excInfo.fault == HardFault) &&
57 (excInfo.origFault != HardFault) then
58 HFSR.FORCED = '1';
59 SetPending(excInfo.fault, excInfo.isSecure, TRUE);
60
61 // If there is a pending exception with sufficient priority take it now. This
62 // is done before committing PC and ITSTATE changes caused by the previous
63 // instruction so that calls to ThisInstrAddr(), NextInstrAddr(),
64 // ThisInstrITState(), NextInstrITState() represent the context the
65 // instruction was executed in. IE so the correct context is pushed to the
66 // stack
67 (takeException, exception, excIsSecure) = PendingExceptionDetails();
68 if takeException then
69 // If a fault occurred during an exception return then the exception
70 // stack frame will already be on the stack, as a result entry to the
71 // next exception is treated as if it were a tail chain.
72 pePriority = ExecutionPriority();
73 peException = UInt(IPSr.Exception);
74 peIsSecure = IsSecure();
75 if excRetFault then
76 // If the fault occurred during ExceptionTaken() then LR will have
77 // been updated with the new exception return value. To excReturn
78 // consistent with the state of the exception stack frame we need to
79 // use the updated version in this case. If no updates have occurred
80 // then the excReturn value from the previous exception return is
81 // used.
82 nextExcReturn = if excInfo.inExcTaken then LR else excReturn;

```

```

82 excInfo = TailChain(exception, excIsSecure, nextExcReturn);
83 else
84 excInfo = ExceptionEntry(exception, excIsSecure, instExecOk);
85 // Handle any derived faults that have occurred
86 if excInfo.fault != NoFault then
87 DerivedLateArrival(pePriority, peException, peIsSecure, excInfo,
88 exception, excIsSecure);
89
90 // If the PC has moved away from the lockup address (eg because an NMI
91 // has been taken) leave the lockup state.
92 if DHCSR.S_LOCKUP == '1' && NextInstrAddr() != 0xEFFFFFFE[31:0] then
93 DHCSR.S_LOCKUP = '0';
94 // Only advance the PC and ITSTATE if not locked up.
95 if DHCSR.S_LOCKUP != '1' then
96 // Commit PC and ITSTATE changes ready for the next instruction.
97 _R[RName_PC] = NextInstrAddr();
98 _PCChanged = FALSE;
99 if HaveMainExt() then
100 EPSR.IT = NextInstrITState();
101 _ITStateChanged = FALSE;

```

### E2.1.170 InstructionSynchronizationBarrier

```

1 // InstructionSynchronizationBarrier()
2 // =====
3 // Perform an instruction synchronization barrier operation
4
5 InstructionSynchronizationBarrier(bits(4) option);

```

### E2.1.171 Int

```

1 // Int()
2 // =====
3
4 integer Int(bits(N) x, boolean unsigned)
5 result = if unsigned then UInt(x) else SInt(x);
6 return result;

```

### E2.1.172 IntegerZeroDivideTrappingEnabled

```

1 // IntegerZeroDivideTrappingEnabled()
2 // =====
3
4 boolean IntegerZeroDivideTrappingEnabled()
5 // DIV_0_TRP bit in CCR is RAZ/WI if Main Extension is not implemented
6 return CCR.DIV_0_TRP == '1';

```

### E2.1.173 IsAccessible

```

1 // IsAccessible()
2 // =====
3
4 (bit, bit, bits(8), boolean) IsAccessible(bits(32) address, boolean forceunpriv,
5 boolean isSecure)
6 bit write;
7 bit read;
8
9 // Work out which privilege level the current mode in the Non-secure state
10 // is subject to
11 if forceunpriv then
12 isPrivileged = FALSE;
13 else
14 isPrivileged = (CurrentMode() == PEMode_Handler) || (if isSecure then
15 CONTROL_S.nPRIV == '0' else CONTROL_NS.nPRIV == '0');

```



```

16 (-, perms) = MPUCheck(address, AccType_NORMAL, isPrivileged, isSecure);
17 if !perms.apValid then
18 write = '0';
19 read = '0';
20 else
21 case perms.ap of
22 when '00' (write, read) = if isPrivileged then ('1','1') else ('0','0');
23 when '01' (write, read) = ('1','1') ;
24 when '10' (write, read) = if isPrivileged then ('0','1') else ('0','0');
25 when '11' (write, read) = ('0','1');
26 return (write, read, perms.region, perms.regionValid);

```

### E2.1.174 IsActiveForState

```

1 // IsActiveForState()
2 // =====
3
4 boolean IsActiveForState(integer exception, boolean isSecure)
5 if !HaveSecurityExt() then
6 isSecure = FALSE;
7 // If the exception is configurable then check which domain it
8 // currently targets. If its not configurable then the active flags can be
9 // used directly.
10 if IsExceptionTargetConfigurable(exception) then
11 active = ((ExceptionActive[exception] != '00') &&
12 (ExceptionTargetsSecure(exception, isSecure) == isSecure));
13 else
14 idx = if isSecure then 0 else 1;
15 active = ExceptionActive[exception][idx] == '1';
16 return active;

```

### E2.1.175 IsAligned

```

1 // IsAligned()
2 // =====
3
4 boolean IsAligned(bits(32) address, integer size)
5 assert size IN {1,2,4,8};
6 mask = (size-1)[31:0]; // integer to bit string conversion
7 return IsZero(address AND mask);

```

### E2.1.176 IsCPEnabled

```

1 // IsCPEnabled()
2 // =====
3
4 (boolean, boolean) IsCPEnabled(integer cp, boolean privileged, boolean secure)
5 // Check Coprocessor Access Control Register for permission to use coprocessor
6 boolean enabled;
7 boolean forceToSecure = FALSE;
8
9 cpacr = if secure then CPACR_S else CPACR_NS;
10 case cpacr[(cp*2)+1:cp*2] of
11 when '00'
12 enabled = FALSE;
13 when '01'
14 enabled = privileged;
15 when '10'
16 UNPREDICTABLE;
17 when '11' // access permitted by CPACR
18 enabled = TRUE;
19
20 if enabled && HaveSecurityExt() then
21 // Check if access in forbidden by NSACR
22 if !secure && NSACR[cp] == '0' then

```

```

23 enabled = FALSE;
24 forceToSecure = TRUE;
25
26 // Check if the coprocessor state unknown flag.
27 if enabled && CPPWR_S[cp*2] == '1' then
28 enabled = FALSE;
29 // Check SUS bit to determine the target state of any fault
30 forceToSecure = CPPWR_S[(cp*2)+1] == '1';
31
32 return (enabled, secure || forceToSecure);
33
34 (boolean, boolean) IsCPEnabled(integer cp)
35 return IsCPEnabled(cp, CurrentModeIsPrivileged(), IsSecure());

```

### E2.1.177 IsCPInstruction

```

1 // IsCPInstruction()
2 // =====
3
4 (boolean, integer) IsCPInstruction(bits(32) instr)
5 isCp = instr IN { '111x1110xxxxxxxxxxxxxxxxxxxxxxxxxxxx',
6 '111x110xxxxxxxxxxxxxxxxxxxxxxxxxxxx' };
7 cpNum = if isCp then UInt(instr[11:8]) else integer UNKNOWN;
8 // CP 11 instructions are treated as CP10
9 if cpNum == 11 then
10 cpNum = 10;
11 return (isCp, cpNum);

```

### E2.1.178 IsDWTConfigUnpredictable

```

1 // IsDWTConfigUnpredictable()
2 // =====
3 // Checks for some of the UNPREDICTABLE cases for various combination of MATCH and
4 // ACTION for each comparator.
5
6 boolean IsDWTConfigUnpredictable(integer N)
7
8 no_trace = (!HaveMainExt() || DWT_CTRL.NOTRCPKT == '1' || !HaveITM());
9
10 // First pass check of MATCH field - coarse checks
11 case DWT_FUNCTION[N].MATCH of
12 when '0000' // Disabled
13 return FALSE;
14 when '0001' // Cycle counter match
15 if !HaveMainExt() || DWT_CTRL.NOCYCNT == '1' || DWT_FUNCTION[N].ID[0] == '0'
16 then
17 return TRUE;
18 when '001x' // Instruction address
19 if (DWT_FUNCTION[N].ID[1] == '0' || DWT_FUNCTION[N].DATAVSIZE != '01' ||
20 DWT_COMP[N][0] == '1') then
21 return TRUE;
22 when '01xx' // Data address
23 lsb = UInt(DWT_FUNCTION[N].DATAVSIZE);
24 if DWT_FUNCTION[N].ID[3] == '0' || (lsb > 0 && !IsZero(DWT_COMP[N][lsb-1:0]))
25 then
26 return TRUE;
27 when '1100', '1101', '1110' // Data address with value
28 if no_trace then return TRUE;
29 lsb = UInt(DWT_FUNCTION[N].DATAVSIZE);
30 if DWT_FUNCTION[N].ID[3] == '0' || (lsb > 0 && !IsZero(DWT_COMP[N][lsb-1:0]))
31 then
32 return TRUE;
33 when '10xx' // Data value
34 Vsize = 2^UInt(DWT_FUNCTION[N].DATAVSIZE);
35 if (!HaveMainExt() || DWT_FUNCTION[N].ID[2] == '0' ||
36 (Vsize != 4 && DWT_COMP[N][31:16] != DWT_COMP[N][15:0]) ||
37 (Vsize == 1 && DWT_COMP[N][15:8] != DWT_COMP[N][7:0])) then

```

```

35 return TRUE;
36 otherwise
37 return TRUE;
38
39 // Second pass MATCH check - linked and limit comparators
40 case DWT_FUNCTION[N].MATCH of
41 when '0011' // Instruction address limit
42 if (N == 0 || DWT_FUNCTION[N].ID[4] == '0' ||
43 DWT_FUNCTION[N-1].MATCH IN {'0001', '0011', '01xx', '1xxx'} ||
44 UInt(DWT_COMP[N]) <= UInt(DWT_COMP[N-1])) then
45 return TRUE;
46 if DWT_FUNCTION[N-1].MATCH == '0000' then return FALSE;
47 when '0111' // Data address limit
48 if (N == 0 || DWT_FUNCTION[N].ID[4] == '0' ||
49 DWT_FUNCTION[N-1].MATCH IN {'0001', '001x', '0111', '10xx'} ||
50 DWT_FUNCTION[N].DATAVSIZE != '00' || DWT_FUNCTION[N-1].DATAVSIZE != '00' ||
51 UInt(DWT_COMP[N]) <= UInt(DWT_COMP[N-1])) then
52 return TRUE;
53 if DWT_FUNCTION[N-1].MATCH == '0000' then return FALSE;
54 when '1011' // Linked data value
55 if (N == 0 || DWT_FUNCTION[N].ID[4] == '0' ||
56 DWT_FUNCTION[N-1].MATCH IN {'0001', '001x', '0111', '10xx'} ||
57 DWT_FUNCTION[N].DATAVSIZE != DWT_FUNCTION[N-1].DATAVSIZE) then
58 return TRUE;
59 if DWT_FUNCTION[N-1].MATCH == '0000' then return FALSE;
60
61 // Check DATAVSIZE is permitted
62 if DWT_FUNCTION[N].DATAVSIZE == '11' then return TRUE;
63
64 // Check the ACTION is allowed for the MATCH type
65 case DWT_FUNCTION[N].ACTION of
66 when '00' // CMPMATCH trigger only
67 if DWT_FUNCTION[N].MATCH IN {'1100', '1101', '1110'} then
68 return TRUE;
69 when '01' // Debug event
70 if DWT_FUNCTION[N].MATCH IN {'0011', '0111', '1100', '1101', '1110'} then
71 return TRUE;
72 when '10' // Data Trace Match or Data Value packet
73 if no_trace || DWT_FUNCTION[N].MATCH IN {'0011', '0111'} then
74 return TRUE;
75 when '11' // Other Data Trace packet
76 if (no_trace || DWT_FUNCTION[N].MATCH IN {'0010', '1000', '1001', '1010'} ||
77 (DWT_FUNCTION[N].MATCH == '0011' && DWT_FUNCTION[N-1].ACTION != '00') ||
78 (DWT_FUNCTION[N].MATCH == '0111' && DWT_FUNCTION[N-1].MATCH == '01xx' &&
79 DWT_FUNCTION[N-1].ACTION IN {'01', '10'}) ||
80 (DWT_FUNCTION[N].MATCH == '0111' && DWT_FUNCTION[N-1].MATCH == '11xx' &&
81 DWT_FUNCTION[N-1].ACTION IN {'00', '01'})) then
82 return TRUE;
83
84 return FALSE; // Passes checks

```

### E2.1.179 IsDWTEnabled

```

1 // IsDWTEnabled()
2 // =====
3 // Check whether DWT is enabled.
4
5 boolean IsDWTEnabled()
6 return HaveDWT() && DEMCR.TRCENA == '1' && ((!IsSecure() && NoninvasiveDebugAllowed()) ||
7 SecureNoninvasiveDebugAllowed());

```

### E2.1.180 IsExceptionTargetConfigurable

```

1 // IsExceptionTargetConfigurable()
2 // =====
3
4 boolean IsExceptionTargetConfigurable(integer e)

```

```

5 if HaveSecurityExt () then
6 case e of
7 when NMI
8 configurable = TRUE;
9 when BusFault
10 configurable = TRUE;
11 when DebugMonitor
12 configurable = TRUE;
13 when SysTick
14 // If there is only 1 SysTick instance then the target domain is
15 // configurable.
16 configurable = HaveSysTick() == 1;
17 otherwise
18 // Exceptions numbers lower than 16 that are not listed in this
19 // function are not configurable in this context.
20 configurable = e >= 16;
21 else
22 configurable = FALSE;
23 return configurable;

```

**E2.1.181 IsExclusiveGlobal**

```

1 // IsExclusiveGlobal
2 // =====
3 // Checks if PE has marked in a global record an address range as "exclusive access
4 // requested" that covers at least the size bytes from address
5
6 boolean IsExclusiveGlobal(bits(32) address, integer processorid, integer size);

```

**E2.1.182 IsExclusiveLocal**

```

1 // IsExclusiveLocal
2 // =====
3 // Checks if PE has marked in a local record an address range as "exclusive access
4 // requested" that covers at least the size bytes from address
5
6 boolean IsExclusiveLocal(bits(32) address, integer processorid, integer size);

```

**E2.1.183 IsIrqValid**

```

1 // IsIrqValid()
2 // =====
3 // Check whether given exception number denotes a valid external interrupt
4 // implemented by PE.
5
6 boolean IsIrqValid(integer e);

```

**E2.1.184 IsOnes**

```

1 // IsOnes()
2 // =====
3
4 boolean IsOnes(bits(N) x)
5 return x == Ones(N);

```

**E2.1.185 IsReqExcPriNeg**

```

1 // IsReqExcPriNeg()
2 // =====
3
4 boolean IsReqExcPriNeg(boolean secure)
5 // This function checks if the requested execution priority is negative for
6 // the specified security domain. That is, NMI or HardFault is active, or

```

```

7 // FAULTMASK is set. It does not take account of AIRCR.PRIS so returns TRUE
8 // if FAULTMASK_NS is set even if PRIS is set to restrict Non-secure priorities
9 // to the range 0x80-0x7E
10
11 neg = (IsActiveForState(NMI, secure) || IsActiveForState(HardFault, secure));
12 if HaveMainExt() then
13 faultmask = if secure then FAULTMASK_S else FAULTMASK_NS;
14 if faultmask.FM == '1' then
15 neg = TRUE;
16 return neg;

```

**E2.1.186 IsSecure**

```

1 // IsSecure()
2 // =====
3
4 boolean IsSecure()
5 return HaveSecurityExt() && CurrentState == SecurityState_Secure;

```

**E2.1.187 IsZero**

```

1 // IsZero()
2 // =====
3
4 boolean IsZero(bits(N) x)
5 return x == Zeros(N);

```

**E2.1.188 IsZeroBit**

```

1 // IsZeroBit()
2 // =====
3
4 bit IsZeroBit(bits(N) x)
5 return if IsZero(x) then '1' else '0';

```

**E2.1.189 ITAdvance**

```

1 // ITAdvance()
2 // =====
3
4 ITAdvance()
5 if ITSTATE[2:0] == '000' then
6 ITSTATE = '00000000';
7 else
8 ITSTATE[4:0] = LSL(ITSTATE[4:0], 1);

```

**E2.1.190 ITSTATE**

```

1 // ITSTATE
2 // =====
3
4 ITSTATEType ITSTATE
5 return ThisInstrITState();
6
7 ITSTATE = ITSTATEType value
8 // Writes to ITSTATE don't take effect immediately, instead they change the
9 // value returned by NextInstrITState().
10 _NextInstrITState = value;
11 _ITStateChanged = TRUE;

```

**E2.1.191 ITSTATETYPE**

```

1 // If-Then execution state bits for the T32 IT instruction.
2
3 type ITSTATEType = bits(8);

```

### E2.1.192 LastInITBlock

```

1 // LastInITBlock()
2 // =====
3
4 boolean LastInITBlock()
5 return (ITSTATE[3:0] == '1000');

```

### E2.1.193 LoadWritePC

```

1 // LoadWritePC()
2 // =====
3
4 LoadWritePC(bits(32) address, integer baseReg, bits(32) baseRegVal, boolean baseRegUpdate,
5 boolean spLimCheck)
6
7 if baseRegUpdate then
8 regName = LookUpRName(baseReg);
9 oldBaseVal = R[baseReg];
10 if spLimCheck then
11 RSPCheck[baseReg] = baseRegVal;
12 else
13 R[baseReg] = baseRegVal;
14
15 // Attempt to update the PC, which may result in a fault
16 excInfo = BXWritePC(address, FALSE);
17
18 if baseRegUpdate && excInfo.fault != NoFault then
19 // Restore the previous base reg value, SP limit checking is not performed
20 _R[regName] = oldBaseVal;
21
22 HandleException(excInfo);

```

### E2.1.194 Lockup

```

1 // Lockup()
2 // =====
3
4 Lockup(boolean termInst)
5 DHCSR.S_LOCKUP = '1';
6 // Branch to the lockup address.
7 BranchToAndCommit(0xEFFFFFFE[31:0]);
8 if termInst then
9 EndOfInstruction();

```

### E2.1.195 LookUpRName

```

1 // LookUpRName()
2 // =====
3
4 RName LookUpRName(integer n)
5 assert n >= 0 && n <= 15;
6 case n of
7 when 0 result = RName0;
8 when 1 result = RName1;
9 when 2 result = RName2;
10 when 3 result = RName3;
11 when 4 result = RName4;
12 when 5 result = RName5;
13 when 6 result = RName6;

```

```

14 when 7 result = RName7;
15 when 8 result = RName8;
16 when 9 result = RName9;
17 when 10 result = RName10;
18 when 11 result = RName11;
19 when 12 result = RName12;
20 when 13 result = LookUpSP();
21 when 14 result = RName_LR;
22 when 15 result = RName_PC;
23 return result;

```

### E2.1.196 LookUpSP

```

1 // LookUpSP()
2 // =====
3
4 RName LookUpSP()
5 return LookUpSP_with_security_mode(IsSecure(), CurrentMode());

```

### E2.1.197 LookUpSP\_with\_security\_mode

```

1 // LookUpSP_with_security_mode()
2 // =====
3
4 RName LookUpSP_with_security_mode(boolean isSecure, PEMode mode)
5 RName sp;
6 bit spSel;
7
8 // Get the SPSEL bit corresponding to the Security state requested
9 if isSecure then
10 spSel = CONTROL_S.SPSEL;
11 else
12 spSel = CONTROL_NS.SPSEL;
13
14 // Should we be using the process or main stack pointers
15 if spSel == '1' && mode == PEMode_Thread then
16 if isSecure then
17 sp = RNameSP_Process_Secure;
18 else
19 sp = RNameSP_Process_NonSecure;
20 else
21 if isSecure then
22 sp = RNameSP_Main_Secure;
23 else
24 sp = RNameSP_Main_NonSecure;
25 return sp;

```

### E2.1.198 LookUpSPLim

```

1 // LookUpSPLim()
2 // =====
3
4 (bits(32), boolean) LookUpSPLim(RName spreg)
5 case spreg of
6 when RNameSP_Main_Secure limit = MSPLIM_S.LIMIT:'000';
7 when RNameSP_Process_Secure limit = PSPLIM_S.LIMIT:'000';
8 when RNameSP_Main_NonSecure
9 limit = if HaveMainExt() then MSPLIM_NS.LIMIT:'000' else Zeros(32);
10 when RNameSP_Process_NonSecure
11 limit = if HaveMainExt() then PSPLIM_NS.LIMIT:'000' else Zeros(32);
12 otherwise
13 assert (FALSE);
14
15 // Check CCR.STKOFHFNMIGN to determine if the limit should actually be
16 // applied. When checking if CCR.STKOFHFNMIGN should apply the requested

```

```

17 // execution priority is considered, and AIRCR.PRIS is ignored.
18 secure = ((spreg == RNameSP_Main_Secure) ||
19 (spreg == RNameSP_Process_Secure));
20 assert (!secure || HaveSecurityExt());
21 if HaveMainExt() && IsReqExcPriNeg(secure) then
22 ignLimit = if secure then CCR_S.STKOFHFNMIGN else CCR_NS.STKOFHFNMIGN;
23 applyLimit = (ignLimit == '0');
24 else
25 applyLimit = TRUE;
26
27 return (limit, applyLimit);

```

**E2.1.199 LowestSetBit**

```

1 // LowestSetBit()
2 // =====
3
4 integer LowestSetBit(bits(N) x)
5 for i = 0 to N-1
6 if x[i] == '1' then return i;
7 return N;

```

**E2.1.200 LR**

```

1 // LR
2 // ==
3
4 // Non-assignment form
5 bits(32) LR
6 return R[14];
7
8 // Assignment form
9
10 LR = bits(32) value
11 R[14] = value;

```

**E2.1.201 LSL**

```

1 // LSL()
2 // =====
3
4 bits(N) LSL(bits(N) x, integer shift)
5 assert shift >= 0;
6 if shift == 0 then
7 result = x;
8 else
9 (result, -) = LSL_C(x, shift);
10 return result;

```

**E2.1.202 LSL\_C**

```

1 // LSL_C()
2 // =====
3
4 (bits(N), bit) LSL_C(bits(N) x, integer shift)
5 assert shift > 0;
6 extended_x = x : Zeros(shift);
7 result = extended_x[N-1:0];
8 carry_out = extended_x[N];
9 return (result, carry_out);

```

**E2.1.203 LSR**



```

1 // LSR()
2 // =====
3
4 bits(N) LSR(bits(N) x, integer shift)
5 assert shift >= 0;
6 if shift == 0 then
7 result = x;
8 else
9 (result, -) = LSR_C(x, shift);
10 return result;

```

### E2.1.204 LSR\_C

```

1 // LSR_C()
2 // =====
3
4 (bits(N), bit) LSR_C(bits(N) x, integer shift)
5 assert shift > 0;
6 extended_x = ZeroExtend(x, shift+N);
7 result = extended_x[shift+N-1:shift];
8 carry_out = extended_x[shift-1];
9 return (result, carry_out);

```

### E2.1.205 MAIRDecode

```

1 // MAIRDecode()
2 // =====
3
4 MemoryAttributes MAIRDecode(bits(8) attrfield, bits(2) sh)
5 // Converts the MAIR attributes to orthogonal attribute and
6 // hint fields.
7 MemoryAttributes memattrs;
8 // Decoding MAIRO/MAIR1 Registers
9 if attrfield[7:4] == '0000' then
10 unpackinner = FALSE;
11 memattrs.memtype = MemType_Device;
12 memattrs.shareable = TRUE;
13 memattrs.outershareable = TRUE;
14 memattrs.innerattrs = bits(2) UNKNOWN;
15 memattrs.outerattrs = bits(2) UNKNOWN;
16 memattrs.innerhints = bits(2) UNKNOWN;
17 memattrs.outerhints = bits(2) UNKNOWN;
18 memattrs.innertransient = boolean UNKNOWN;
19 memattrs.outertransient = boolean UNKNOWN;
20 case attrfield[3:0] of
21 when '0000' memattrs.device = DeviceType_nGnRnE;
22 when '0100' memattrs.device = DeviceType_nGnRE;
23 when '1000' memattrs.device = DeviceType_nGRE;
24 when '1100' memattrs.device = DeviceType_GRE;
25 if attrfield[1:0] != '00' then UNPREDICTABLE;
26 else
27 unpackinner = TRUE;
28 memattrs.memtype = MemType_Normal;
29 memattrs.device = DeviceType UNKNOWN;
30 memattrs.outerhints = attrfield[5:4];
31 memattrs.shareable = sh[1] == '1';
32 memattrs.outershareable = sh == '10';
33 if sh == '01' then UNPREDICTABLE;
34
35 if attrfield[7:6] == '00' then
36 memattrs.outerattrs = '10';
37 memattrs.outertransient = TRUE;
38 elsif attrfield[7:6] == '01' then
39 if attrfield[5:4] == '00' then
40 memattrs.outerattrs = '00';
41 memattrs.outertransient = FALSE;
42 else

```

```

43 memattrs.outerattrs = '11';
44 memattrs.outertransient = TRUE;
45 else
46 memattrs.outerattrs = attrfield[7:6];
47 memattrs.outertransient = FALSE;
48 if unpackinner then
49 if attrfield[3:0] == '0000' then UNPREDICTABLE;
50 else
51 if attrfield[3:2] == '00' then
52 memattrs.innerattrs = '10';
53 memattrs.innerhints = attrfield[1:0];
54 memattrs.innertransient = TRUE;
55 elsif attrfield[3:2] == '01' then
56 memattrs.innerhints = attrfield[1:0];
57 if attrfield[1:0] == '00' then
58 memattrs.innerattrs = '00';
59 memattrs.innertransient = FALSE;
60 else
61 memattrs.innerattrs = '11';
62 memattrs.innertransient = TRUE;
63 elsif attrfield[3:2] == '10' then
64 memattrs.innerhints = attrfield[1:0];
65 memattrs.innerattrs = '10';
66 memattrs.innertransient = FALSE;
67 elsif attrfield[3:2] == '11' then
68 memattrs.innerhints = attrfield[1:0];
69 memattrs.innerattrs = '11';
70 memattrs.innertransient = FALSE;
71 else UNPREDICTABLE;
72 return memattrs;

```

### E2.1.206 MarkExclusiveGlobal

```

1 // MarkExclusiveGlobal
2 // =====
3 // Records in a global record that PE has requested "exclusive access" covering
4 // at least size bytes from the address
5
6 MarkExclusiveGlobal(bits(32) address, integer processorid, integer size);

```

### E2.1.207 MarkExclusiveLocal

```

1 // MarkExclusiveLocal
2 // =====
3 // Records in a local record that PE has requested "exclusive access" covering
4 // at least size bytes from the address.
5
6 MarkExclusiveLocal(bits(32) address, integer processorid, integer size);

```

### E2.1.208 Max

```

1 // Max()
2 // =====
3
4 integer Max(integer a, integer b)
5 return if a >= b then a else b;
6
7 real Max(real a, real b)
8 return if a >= b then a else b;

```

### E2.1.209 MaxExceptionNum

```

1 // MaxExceptionNum()
2 // =====

```

```

3 // Returns the maximum exception number supported
4
5 integer MaxExceptionNum()
6 if HaveMainExt() then
7 return 511;
8 else
9 return 47;

```

### E2.1.210 MemA

```

1 // MemA[]
2 // =====
3
4 bits(8*size) MemA[bits(32) address, integer size]
5 return MemA_with_priv[address, size, FindPriv(), TRUE];
6
7 MemA[bits(32) address, integer size] = bits(8*size) value
8 MemA_with_priv[address, size, FindPriv(), TRUE] = value;
9 return;

```

### E2.1.211 MemA\_with\_priv

```

1 // MemA_with_priv[]
2 // =====
3
4 // Non-assignment form
5
6 bits(8*size) MemA_with_priv[bits(32) address, integer size, boolean privileged,
7 boolean aligned]
8 (excInfo, value) = MemA_with_priv_security(address, size, AccType_NORMAL,
9 privileged, IsSecure(), aligned);
10 HandleException(excInfo);
11 return value;
12
13
14 // Assignment form
15
16 MemA_with_priv[bits(32) address, integer size, boolean privileged,
17 boolean aligned] = bits(8*size) value
18 excInfo = MemA_with_priv_security(address, size, AccType_NORMAL, privileged,
19 IsSecure(), aligned, value);
20 HandleException(excInfo);

```

### E2.1.212 MemA\_with\_priv\_security

```

1 // MemA_with_priv_security()
2 // =====
3
4 // Non-assignment form
5
6 (ExcInfo, bits(8*size)) MemA_with_priv_security(bits(32) address, integer size,
7 AccType acctype, boolean privileged,
8 boolean secure, boolean aligned)
9
10 // Check alignment
11 excInfo = DefaultExcInfo();
12 if !IsAligned(address, size) then
13 if HaveMainExt() then
14 UFSR.UNALIGNED = '1';
15 // Create the exception. NOTE: If Main Extension is not implemented the fault
16 // always escalates to a HardFault
17 excInfo = CreateException(UsageFault, TRUE, secure);
18
19 // Check permissions and get attributes
20 if excInfo.fault == NoFault then
21 (excInfo, memaddrdesc) = ValidateAddress(address, acctype, privileged, secure,

```

```

21 FALSE, aligned);
22
23 if excInfo.fault == NoFault then
24 // Memory array access, and sort out endianness
25 (error, value) = _Mem(memaddrdesc, size);
26
27 // Check if a synchronous BusFault occurred, async BusFaults are handled
28 // in RaiseAsyncBusFault()
29 if error then
30 value = bits(8*size) UNKNOWN;
31 if HaveMainExt() then
32 if acctype == AccType_STACK then
33 BFSR.UNSTKERR = '1';
34 elseif acctype IN {AccType_NORMAL, AccType_ORDERED} then
35 BFAR.ADDRESS = address;
36 BFSR.BFARVALID = '1';
37 BFSR.PRECISERR = '1';
38
39 // Generate BusFault exception if it cannot be ignored.
40 if !IsReqExcPriNeg(secure) || (CCR.BFHFNMIGN == '0') then
41 // Create the exception. NOTE: If Main Extension is not implemented
42 // the fault always escalates to a HardFault
43 excInfo = CreateException(BusFault, FALSE, boolean UNKNOWN);
44
45 // Check for Watch Point Match
46 elseif IsDWTEnabled() then
47 bits(32) dvalue = ZeroExtend(value);
48 DWT_DataMatch(address, size, dvalue, TRUE, secure);
49
50 // PPB (0xE0000000 to 0xE00FFFFFF) is always little endian
51 if AIRCR.ENDIANNESS == '1' && UInt(address[31:20]) != 0xE00 then
52 value = BigEndianReverse(value, size);
53
54 return (excInfo, value);
55
56 // Assignment form
57
58 ExcInfo MemA_with_priv_security(bits(32) address, integer size, AccType acctype,
59 boolean privileged, boolean secure, boolean aligned,
60 bits(8*size) value)
61
62 // Check alignment
63 excInfo = DefaultExcInfo();
64 if !IsAligned(address, size) then
65 if HaveMainExt() then
66 UFSR.UNALIGNED = '1';
67 // Create the exception. NOTE: If Main Extension is not implemented the fault
68 // always escalates to a HardFault
69 excInfo = CreateException(UsageFault, TRUE, secure);
70
71 // Check permissions and get attributes
72 if excInfo.fault == NoFault then
73 (excInfo, memaddrdesc) = ValidateAddress(address, acctype, privileged, secure,
74 TRUE, aligned);
75
76 if excInfo.fault == NoFault then
77 // Effect on exclusives
78 if memaddrdesc.memattrs.shareable then
79 ClearExclusiveByAddress(memaddrdesc.address,
80 ProcessorID(), size); // see Note
81
82 // Sort out endianness, then memory array access
83 // PPB (0xE0000000 to 0xE0100000) is always little endian
84 if AIRCR.ENDIANNESS == '1' && UInt(address[31:20]) != 0xE00 then
85 value = BigEndianReverse(value, size);
86
87 // Check for Watch Point Match
88 if IsDWTEnabled() then
89 bits(32) dvalue = ZeroExtend(value);
90 DWT_DataMatch(address, size, dvalue, FALSE, secure);

```

```

90
91 if _Mem(memaddrdesc, size, value) then
92 // Synchronous BusFault occurred. NOTE: async BusFaults are handled
93 // in RaiseAsyncBusFault()
94
95 // Check whether the execution priority is negative.
96 // If the access is due to lazy FP state preservation the FPCCR flag
97 // indicating whether a HardFault could be taken is used to determine if the
98 // priority should be considered to be negative rather than the current
99 // execution priority.
100 if acctype == AccType_LAZYFP then
101 negativePri = FPCCR_S.HFRDY == '0';
102 else
103 negativePri = IsReqExcPriNeg(secure);
104
105 if HaveMainExt() then
106 if acctype == AccType_STACK then
107 BFSR.STKERR = '1';
108 elseif acctype == AccType_LAZYFP then
109 BFSR.LSPERR = '1';
110 elseif acctype IN {AccType_NORMAL, AccType_ORDERED} then
111 BFAR.ADDRESS = address;
112 BFSR.BFARVALID = '1';
113 BFSR.PRECISERR = '1';
114
115 // Generate BusFault exception if it cannot be ignored.
116 if !negativePri || (CCR.BFHFNMIGN == '0') then
117 // Create the exception. NOTE: If Main Extension is not implemented
118 // the fault always escalates to a HardFault
119 excInfo = CreateException(BusFault, FALSE, boolean UNKNOWN);
120 return excInfo;

```

### E2.1.213 MemI

```

1 // MemI()
2 // =====
3
4 bits(16) MemI[bits(32) address]
5 // Check permissions and get attributes
6 // NOTE: The privilege flag passed to ValidateAddress may be overridden if
7 // the security of the memory is different from the current security
8 // state, eg when performing a Non-secure to Secure function call.
9 (excInfo, memaddrdesc) = ValidateAddress(address, AccType_IFETCH, FindPriv(),
10 IsSecure(), FALSE, TRUE);
11
12 if excInfo.fault == NoFault then
13 (error, value) = _Mem(memaddrdesc, 2);
14 if error then
15 value = bits(16) UNKNOWN;
16 BFSR.IBUSERR = '1';
17 // Create the exception. NOTE: If Main Extension is not implemented the fault
18 // always escalates to a HardFault
19 excInfo = CreateException(BusFault, FALSE, boolean UNKNOWN);
20 HandleException(excInfo);
21 if IsDWTEnabled() then DWT_InstructionMatch(address);
22 return value;

```

### E2.1.214 MemO

```

1 // MemO[] - non-assignment form
2 // =====
3
4 bits(8*size) MemO[bits(32) address, integer size]
5 (excInfo, value) = MemA_with_priv_security(address, size, AccType_ORDERED,
6 FindPriv(), IsSecure(), TRUE);
7 HandleException(excInfo);
8 return value;
9

```

```

10
11 // MemO[] - assignment form
12 // =====
13
14 MemO[bits(32) address, integer size] = bits(8*size) value
15 excInfo = MemA_with_priv_security(address, size, AccType_ORDERED, FindPriv(),
16 IsSecure(), TRUE, value);
17 HandleException(excInfo);

```

### E2.1.215 MemoryAttributes

```

1 // v8-M Memory Attributes
2 type MemoryAttributes is (
3 MemType memtype,
4 DeviceType device, // For Device memory
5 bits(2) innerattrs, // The possible encodings for each attributes field are:
6 bits(2) outerattrs, // '00' = Non-cacheable; '01' = RESERVED
7 // '10' = Write-Through; '11' = Write-Back
8 bits(2) innerhints, // The possible encodings for the hints are as follows
9 bits(2) outerhints, // '00' = No-Allocate; '01' = Write-Allocate
10 // '10' = Read-Allocate; ;'11' = Read-Allocate and Write-Allocate
11 boolean NS, // TRUE if Non-secure, else FALSE
12 boolean innertransient,
13 boolean outertransient,
14 boolean shareable,
15 boolean outershareable
16)

```

### E2.1.216 MemType

```

1 // Types of memory
2
3 enumeration MemType {MemType_Normal, MemType_Device};

```

### E2.1.217 MemU

```

1 // MemU[]
2 // =====
3
4 // Non-assignment form, used for memory reads
5 // =====
6
7 bits(8*size) MemU[bits(32) address, integer size]
8 if HaveMainExt() then
9 return MemU_with_priv[address, size, FindPriv()];
10 else
11 return MemA[address, size];
12
13
14 // Assignment form, used for memory writes
15 // =====
16
17 MemU[bits(32) address, integer size] = bits(8*size) value
18 if HaveMainExt() then
19 MemU_with_priv[address, size, FindPriv()] = value;
20 else
21 MemA[address, size] = value;
22 return;

```

### E2.1.218 MemU\_unpriv

```

1 // MemU_unpriv[]
2 // =====
3

```

```

4 bits(8*size) MemU_unpriv[bits(32) address, integer size]
5 return MemU_with_priv[address, size, FALSE];
6
7 MemU_unpriv[bits(32) address, integer size] = bits(8*size) value
8 MemU_with_priv[address, size, FALSE] = value;
9 return;

```

### E2.1.219 MemU\_with\_priv

```

1 // MemU_with_priv[]
2 // =====
3 // Due to single-copy atomicity constraints, the aligned accesses are distinguished from
4 // the unaligned accesses:
5 // * aligned accesses are performed at their size
6 // * unaligned accesses are expressed as a set of bytes.
7
8 // Non-assignment form
9
10 bits(8*size) MemU_with_priv[bits(32) address, integer size, boolean privileged]
11
12 bits(8*size) value;
13 // Do aligned access, take alignment fault, or do sequence of bytes
14 if address == Align(address, size) then
15 value = MemA_with_priv[address, size, privileged, TRUE];
16 elseif CCR.UNALIGN_TRP == '1' then
17 UFSR.UNALIGNED = '1';
18 excInfo = CreateException(UsageFault, FALSE, boolean UNKNOWN);
19 HandleException(excInfo);
20 else // if unaligned access
21 for i = 0 to size-1
22 value[8*i+7:8*i] = MemA_with_priv[address+i, 1, privileged, FALSE];
23 // PPB (0xE0000000 to 0xE0100000) is always little endian
24 if AIRCR.ENDIANNESS == '1' && UInt(address[31:20]) != 0xE00 then
25 value = BigEndianReverse(value, size);
26
27 return value;
28
29 // Assignment form
30
31 MemU_with_priv[bits(32) address, integer size, boolean privileged] = bits(8*size) value
32
33 // Do aligned access, take alignment fault, or do sequence of bytes
34 if address == Align(address, size) then
35 MemA_with_priv[address, size, privileged, TRUE] = value;
36 elseif CCR.UNALIGN_TRP == '1' then
37 UFSR.UNALIGNED = '1';
38 excInfo = CreateException(UsageFault, FALSE, boolean UNKNOWN);
39 HandleException(excInfo);
40 else // if unaligned access
41 // PPB (0xE0000000 to 0xE0100000) is always little endian
42 if AIRCR.ENDIANNESS == '1' && UInt(address[31:20]) != 0xE00 then
43 value = BigEndianReverse(value, size);
44 for i = 0 to size-1
45 MemA_with_priv[address+i, 1, privileged, FALSE] = value[8*i+7:8*i];
46
47 return;

```

### E2.1.220 MergeExcInfo

```

1 // MergeExcInfo()
2 // =====
3
4 ExcInfo MergeExcInfo(ExcInfo a, ExcInfo b)
5 // The ExcInfo structure is used to determine which exception should be
6 // taken, and how it should be handled (mainly in the case of derived
7 // exceptions).
8 if (b.fault == NoFault) || (a.isTerminal && !b.isTerminal) then

```

```

9 exc = a;
10 elseif (a.fault == NoFault) || (b.isTerminal && !a.isTerminal) then
11 exc = b;
12 elseif (a.fault == b.fault) && (a.isSecure == b.isSecure) then
13 exc = a;
14 else
15 // Propagate the fault with the highest priority (lowest numerical
16 // value).
17 aPri = ExceptionPriority(a.fault, a.isSecure, FALSE);
18 bPri = ExceptionPriority(b.fault, b.isSecure, FALSE);
19
20 // Compare the exception priority values. Exception with the highest priority, which
21 // is the lowest numerical value, is taken and the other exception may be pended.
22 if aPri < bPri then
23 exc = a;
24 pend = b;
25 elseif bPri < aPri then
26 exc = b;
27 pend = a;
28 // If both priority values are equal, the exception numbers are compared.
29 // The exception with the lowest exception number is taken and the other
30 // exception may be pended.
31 elseif a.fault < b.fault then
32 exc = a;
33 pend = b;
34 elseif b.fault < a.fault then
35 exc = b;
36 pend = a;
37 // If the two exception number are equal, the Secure exception is taken and the
38 // Non-secure exception may be pended.
39 elseif a.isSecure && !b.isSecure then
40 exc = a;
41 pend = b;
42 // In any other case exception (b) is taken and exception (a) is pended.
43 else
44 exc = b;
45 pend = a;
46
47 // It is IMPLEMENTATION_DEFINED whether all exceptions generated are visible or not.
48 // If visible, the highest priority exception will become active and lower priority
49 // exceptions will get pended.
50 if boolean IMPLEMENTATION_DEFINED "Overridden exceptions pended" then
51 SetPending(pend.fault, pend.isSecure, TRUE);
52 return exc;

```

### E2.1.221 Min

```

1 // Min()
2 // =====
3
4 integer Min(integer a, integer b)
5 return if a <= b then a else b;
6
7 real Min(real a, real b)
8 return if a <= b then a else b;

```

### E2.1.222 MPUCheck

```

1 // MPUCheck()
2 // =====
3
4 (MemoryAttributes, Permissions) MPUCheck(bits(32) address, AccType acctype,
5 boolean ispriv, boolean secure)
6
7 assert(HaveSecurityExt() || !secure);
8 MemoryAttributes attributes;
9 Permissions perms;

```



```

10 attributes = DefaultMemoryAttributes(address);
11 perms = DefaultPermissions(address);
12 // assume no valid MPU region and not using default memory map
13 hit = FALSE;
14 isPPBaccess = (address[31:20] == '111000000000');
15
16 // Get the MPU registers for the correct security domain
17 if secure then
18 mpu_ctrl = MPU_CTRL_S;
19 mpu_type = MPU_TYPE_S;
20 mair = MPU_MAIR1_S:MPU_MAIRO_S;
21 else
22 mpu_ctrl = MPU_CTRL_NS;
23 mpu_type = MPU_TYPE_NS;
24 mair = MPU_MAIR1_NS:MPU_MAIRO_NS;
25
26 // Pre-compute if the execution priority is negative, as this can affect the
27 // MPU permissions used. NOTE: If Non-secure FAULTMASK is set this is also
28 // considered to be a negative priority for the purpose of the Non-secure
29 // MPU permissions regardless of how Non-secure exceptions are prioritised
30 // with respect to the Secure state.
31 // If the access is due to lazy FP state preservation the FPCCR flag
32 // indicating whether a HardFault could be taken is used to determine if the
33 // priority should be considered to be negative rather than the current
34 // execution priority.
35 if acctype == AccType_LAZYFP then
36 negativePri = FPCCR_S.HFRDY == '0';
37 else
38 negativePri = IsReqExcPriNeg(secure);
39
40 // Determine what MPU permissions should apply based on access type and MPU
41 // configuration
42 if (acctype == AccType_VECTABLE) || isPPBaccess then
43 hit = TRUE; // use default map for PPB and vector table lookups
44 elseif mpu_ctrl.ENABLE == '0' then
45 if mpu_ctrl.HFNMIENA == '1' then UNPREDICTABLE;
46 else hit = TRUE; // always use default map if MPU disabled
47 elseif mpu_ctrl.HFNMIENA == '0' && negativePri then
48 hit = TRUE; // optionally use default for HardFault, NMI and FAULTMASK.
49 else // MPU is enabled so check each individual region
50 if (mpu_ctrl.PRIVDEFENA == '1') && ispriv then
51 hit = TRUE; // optional default as background for Privileged accesses
52
53 regionMatched = FALSE;
54 for r = 0 to (UInt(mpu_type.DREGION) - 1)
55
56 if secure then
57 rbar = __MPU_RBAR_S[r];
58 rlar = __MPU_RLAR_S[r];
59 else
60 rbar = __MPU_RBAR_NS[r];
61 rlar = __MPU_RLAR_NS[r];
62
63 // MPU region enabled so perform checks
64 if rlar.EN == '1' then
65 if ((UInt(address) >= UInt(rbar.BASE : '0000')) &&
66 (UInt(address) <= UInt(rlar.LIMIT : '1111'))) then
67
68
69 // flag error if multiple regions match
70 if regionMatched then
71 perms.regionValid = FALSE;
72 perms.region = Zeros(8);
73 hit = FALSE;
74 else
75 regionMatched = TRUE;
76 perms.ap = rbar.AP;
77 perms.xn = rbar.XN;
78 perms.region = r[7:0];

```

```

79 perms.regionValid = TRUE;
80 hit = TRUE;
81 sh = rbar.SH;
82
83 // parsing MAIRO/1 Register fields
84 index = UInt(rlar.AttrIdx);
85 attrfield = mair[8*index+7:8*index];
86 // decoding MAIRO/1 field and populating memory attributes
87 attributes = MAIRDecode(attrfield, sh);
88
89 if address[31:29] == '111' then // enforce System space execute never
90 perms.xn = '1';
91 if !hit then // Access not allowed if no MPU match and use of default not enabled
92 perms.apValid = FALSE;
93 return (attributes, perms);

```

### E2.1.223 NextInstrAddr

```

1 // NextInstrAddr()
2 // =====
3
4 bits(32) NextInstrAddr()
5 if _PCChanged then
6 return _NextInstrAddr;
7 else
8 return ThisInstrAddr() + ThisInstrLength();

```

### E2.1.224 NextInstrITState

```

1 // NextInstrITState()
2 // =====
3
4 ITSTATEType NextInstrITState()
5 if HaveMainExt() then
6 // If the IT state has been directly modified return that value as the
7 // next state, otherwise advance the IT state normally.
8 if _ITStateChanged then
9 nextState = _NextInstrITState;
10 else
11 nextState = ThisInstrITState();
12 if nextState[2:0] == '000' then
13 nextState = '00000000';
14 else
15 nextState[4:0] = LSL(nextState[4:0], 1);
16 else
17 nextState = Zeros(8);
18 return nextState;

```

### E2.1.225 NoninvasiveDebugAllowed

```

1 // NoninvasiveDebugAllowed()
2 // =====
3
4 boolean NoninvasiveDebugAllowed()
5 return ExternalNoninvasiveDebugEnabled() || HaltingDebugAllowed();

```

### E2.1.226 Ones

```

1 // Ones()
2 // =====
3
4 bits(N) Ones(integer N)
5 return Replicate('1', N);
6

```

```

7 bits(N) Ones()
8 return Ones(N);

```

### E2.1.227 PC

```

1 // PC - non-assignment form
2 // =====
3 bits(32) PC
4 return R[15];

```

### E2.1.228 PEmode

```

1 // The PE execution modes.
2
3 enumeration PEmode {PEmode_Thread, PEmode_Handler};

```

### E2.1.229 PendingExceptionDetails

```

1 // PendingExceptionDetails
2 // =====
3 // Determines whether to take a pending exception or not. This is done based
4 // on current execution priority and the priority of pending exceptions that
5 // are not masked by DHCSR.C_MASKINTS.
6 // Returns whether any pending exception is to be taken, and, if so, the
7 // exception number for the highest priority unmasked exception, and
8 // whether this exception is Secure.
9
10 (boolean, integer, boolean) PendingExceptionDetails();

```

### E2.1.230 PendReturnOperation

```

1 // PendReturnOperation()
2 // =====
3
4 PendReturnOperation(bits(32) returnValue)
5 _NextInstrAddr = returnValue;
6 _PCChanged = TRUE;
7 _PendingReturnOperation = TRUE;
8 return;

```

### E2.1.231 Permissions

```

1 // Access permissions descriptor
2
3 type Permissions is (
4 boolean apValid, // TRUE when ap is valid, else FALSE
5 bits(2) ap, // Access Permission bits, if valid
6 bit xn, // Execute Never bit
7 boolean regionValid, // TRUE if the region number is valid, else FALSE
8 bits(8) region // The MPU region number, if valid
9)

```

### E2.1.232 PopStack

```

1 // PopStack()
2 // =====
3
4 ExcInfo PopStack(EXC_RETURN_Type excReturn)
5 // NOTE: All stack accesses are performed as Unprivileged accesses if
6 // returning to Thread mode and CONTROL.nPRIV is 1 for the destination
7 // Security state.

```

```

8 mode = if excReturn.Mode == '1' then PEMode_Thread else PEMode_Handler;
9 toSecure = HaveSecurityExt() && excReturn.S == '1';
10 spName = LookUpSP_with_security_mode(toSecure, mode);
11 frameptr = _SP(spName);
12 if !IsAligned(frameptr, 8) then UNPREDICTABLE;
13
14 // only stack locations, not the load order, are architected
15
16 // Pop the callee saved registers, when returning from a Non-secure exception
17 // or a Secure one that followed a Non-secure one and therefore still has
18 // the callee register state on the stack.
19 exc = DefaultExcInfo();
20 if toSecure && (excReturn.ES == '0' ||
21 excReturn.DCRS == '0') then
22 // Check the integrity signature, and if so is it correct
23 expectedSig = 0xFEFA125B[31:0];
24 if HaveFPEExt() then
25 expectedSig[0] = excReturn.FType;
26 (exc, integritySig) = Stack(frameptr, 0x0, spName, mode);
27 if exc.fault == NoFault && integritySig != expectedSig then
28 if HaveMainExt() then
29 SFSR.INVIS = '1';
30 // Create the exception. NOTE: If Main Extension is not implemented the fault
31 // always escalates to a HardFault
32 return CreateException(SecureFault, TRUE, TRUE);
33
34 if exc.fault == NoFault then (exc, R[4]) = Stack(frameptr, 0x8, spName, mode);
35 if exc.fault == NoFault then (exc, R[5]) = Stack(frameptr, 0xC, spName, mode);
36 if exc.fault == NoFault then (exc, R[6]) = Stack(frameptr, 0x10, spName, mode);
37 if exc.fault == NoFault then (exc, R[7]) = Stack(frameptr, 0x14, spName, mode);
38 if exc.fault == NoFault then (exc, R[8]) = Stack(frameptr, 0x18, spName, mode);
39 if exc.fault == NoFault then (exc, R[9]) = Stack(frameptr, 0x1C, spName, mode);
40 if exc.fault == NoFault then (exc, R[10]) = Stack(frameptr, 0x20, spName, mode);
41 if exc.fault == NoFault then (exc, R[11]) = Stack(frameptr, 0x24, spName, mode);
42 frameptr = frameptr + 0x28;
43
44 // Unstack the caller saved regs, possibly including the FP regs
45 RETPSR_Type psr;
46 if exc.fault == NoFault then (exc, R[0]) = Stack(frameptr, 0x0, spName, mode);
47 if exc.fault == NoFault then (exc, R[1]) = Stack(frameptr, 0x4, spName, mode);
48 if exc.fault == NoFault then (exc, R[2]) = Stack(frameptr, 0x8, spName, mode);
49 if exc.fault == NoFault then (exc, R[3]) = Stack(frameptr, 0xC, spName, mode);
50 if exc.fault == NoFault then (exc, R[12]) = Stack(frameptr, 0x10, spName, mode);
51 if exc.fault == NoFault then (exc, LR) = Stack(frameptr, 0x14, spName, mode);
52 if exc.fault == NoFault then (exc, pc) = Stack(frameptr, 0x18, spName, mode);
53 if exc.fault == NoFault then (exc, psr) = Stack(frameptr, 0x1C, spName, mode);
54 BranchToAndCommit(pc);
55
56 // Check the XPSR value that has been unstacked is consistent with the mode
57 // being returned to
58 excNum = UInt(psr.Exception);
59 if (exc.fault == NoFault) &&
60 (mode == PEMode_Handler) == (excNum == 0) then
61 if HaveMainExt() then
62 UFSR.INVPC = '1';
63 // Create the exception. NOTE: If Main Extension is not implemented the fault
64 // always escalates to a HardFault
65 return CreateException(UsageFault, FALSE, boolean UNKNOWN);
66 // The IPSR value is set as UNKNOWN if the unstacked IPSR value is not supported by the
67 // PE
68 validIPSR = excNum IN {0, 1, NMI, HardFault, SVCcall, PendSV, SysTick};
69 if !validIPSR && HaveMainExt() then
70 validIPSR = excNum IN {MemManage, BusFault, UsageFault, SecureFault, DebugMonitor};
71
72 // Check also whether excNum is an external interrupt supported by PE
73 if !validIPSR && !IsIrqValid(excNum) then
74 psr.Exception = bits(9) UNKNOWN;
75
76 if HaveFPEExt() then

```

```

76 if excReturn.FType == '0' then
77 // Raise a fault and skip Floating-point operations if requested to expose
78 // Secure Floating-point state to the Non-secure code.
79 if !toSecure && FPCCR_S.LSPACT == '1' then
80 SFSR.LSERR = '1';
81 newExc = CreateException(SecureFault, TRUE, TRUE);
82 // It is IMPLEMENTATION DEFINED whether a MemFault is dropped if
83 // a SecureFault is generated subsequently. If the MemFault is
84 // not dropped the exceptions will be taken based on exception
85 // priority as described in MergeExcInfo()
86 if boolean IMPLEMENTATION_DEFINED "Drop previously generated exceptions" then
87 exc = newExc;
88 else
89 exc = MergeExcInfo(exc, newExc);
90 else
91 lspact = if toSecure then FPCCR_S.LSPACT else FPCCR_NS.LSPACT;
92 if lspact == '1' then // state in FP is still valid
93 if exc.fault == NoFault then
94 if toSecure then
95 FPCCR_S.LSPACT = '0';
96 else
97 FPCCR_NS.LSPACT = '0';
98 else
99 if exc.fault == NoFault then
100 nPriv = if toSecure then CONTROL_S.nPRIV else CONTROL_NS.nPRIV;
101 isPriv = mode == PMode_Handler || nPriv == '0';
102 exc = CheckCPEnabled(10, isPriv, toSecure);
103
104 // If an implementation abandons the unstacking of the Floating-point
105 // Extension registers and to tail chain into a fault or late arriving
106 // interrupt it must clear any Floating-point registers that
107 // would have been unstacked.
108 // NOTE: The requirement to clear the registers only applies
109 // to implementations that include the Security Extensions.
110 // The Floating-point Extension registers that would have been unstacked
111 // become
112 // UNKNOWN in implementations that do not include the
113 // Security Extensions.
114 if exc.fault == NoFault then
115 for i = 0 to 15
116 if exc.fault == NoFault then
117 offset = 0x20+(4*i);
118 (exc, S[i]) = Stack(frameptr, offset, spName, mode);
119 if exc.fault == NoFault then
120 (exc, FPSCR) = Stack(frameptr, 0x60, spName, mode);
121 if toSecure && FPCCR_S.TS == '1' then
122 for i = 0 to 15
123 if exc.fault == NoFault then
124 offset = 0x68+(4*i);
125 (exc, S[i+16]) = Stack(frameptr, offset, spName, mode);
126 if exc.fault != NoFault then
127 for i = 16 to 31
128 S[i] = if HaveSecurityExt() then Zeros(32) else bits(32)
129 UNKNOWN;
130 if exc.fault != NoFault then
131 for i = 0 to 15
132 S[i] = if HaveSecurityExt() then Zeros(32) else bits(32)
133 UNKNOWN;
134 FPSCR = if HaveSecurityExt() then Zeros(32) else bits(32)
135 UNKNOWN;
136 CONTROL.FPCA = NOT(excReturn.FType);
137
138 // If there was not a fault then move the stack pointer to consume the
139 // exception stack frame. NOTE: If a exception return fault occurs and
140 // results in a lockup the stack pointer is updated. This special case is
141 // handled at the point lockup is entered and not here.
142 if exc.fault == NoFault then
143 ConsumeExcStackFrame(excReturn, psr.SPALIGN);

```

```

141
142 if HaveDSPExt () then
143 APSR.GE = psr.GE;
144
145 if IsSecure () then
146 CONTROL_S.SFPA = psr.SFPA;
147
148 IPSR.Exception = psr.Exception; // Load valid IPSR bits from memory
149 EPSR.T = psr.T; // Load valid EPSR bits from memory
150 if HaveMainExt () then
151 APSR[31:27] = psr[31:27]; // Load valid APSR bits from memory
152 SetITSTATEAndCommit (psr.IT); // Load valid ITSTATE from memory
153 else
154 APSR[31:28] = psr[31:28]; // Load valid APSR bits from memory
155 return exc;

```

### E2.1.233 PreserveFPState

```

1 // PreserveFPState()
2 // =====
3
4 PreserveFPState ()
5 // Preserve FP state using address, privilege and relative
6 // priorities recorded during original stacking. Derived
7 // exceptions are handled by TakePreserveFPException().
8
9 // The checks usually performed for stacking using ValidateAddress()
10 // are performed, with the value of ExecutionPriority()
11 // overridden by -1 if FPCCR.HFRDY == '0'.
12
13 isSecure = FPCCR_S.S == '1';
14 if isSecure then
15 ispriv = FPCCR_S.USER == '0';
16 splimviol = FPCCR_S.SPLIMVIOL == '1';
17 fpcar = FPCCR_S;
18 else
19 ispriv = FPCCR_NS.USER == '0';
20 splimviol = FPCCR_NS.SPLIMVIOL == '1';
21 fpcar = FPCCR_NS;
22
23 // Check if the background context had access to the FPU
24 excInfo = CheckCPEnabled(10, ispriv, isSecure);
25
26 // Only perform the memory accesses if the stack limit hasn't been violated
27 if !splimviol then
28
29 // Whether these stores are interruptible is IMPLEMENTATION DEFINED.
30 // Only the stack locations, not the store order, are architected.
31 for i = 0 to 15
32 if excInfo.fault == NoFault then
33 addr = fpcar + (4*i);
34 excInfo = MemA_with_priv_security (addr, 4, AccType_LAZYFP, ispriv, isSecure, TRUE,
35 S[i]);
36
37 if excInfo.fault == NoFault then
38 addr = fpcar + 0x40;
39 excInfo = MemA_with_priv_security (addr, 4, AccType_LAZYFP, ispriv, isSecure, TRUE,
40 FPSCR);
41
42 if isSecure && FPCCR_S.TS == '1' then
43 for i = 0 to 15
44 if excInfo.fault == NoFault then
45 addr = fpcar + (4*i) + 0x48;
46 excInfo = MemA_with_priv_security (addr, 4, AccType_LAZYFP, ispriv, TRUE,
47 TRUE, S[i+16]);
48
49 // If a fault was raised handle it now. This function may call
50 // EndOfInstruction(), as a result any code after this call may not execute.

```

```

49 if excInfo.fault != NoFault then
50 TakePreserveFPException(excInfo);
51
52 // If the stores are interrupted, the register content and LSPACT remain unchanged.
53
54 // If exception with sufficient priority to pre-empt current instruction execution
55 // is raised during FP state preserve, then TakePreserveFPException() will terminate
56 // the current instruction by calling EndOfInstruction().
57 // If the exception results in a lockup state, then TakePreserveFPException() will
58 // enter the lockup state by calling Lockup().
59 // In both above cases where execution of current instruction is not completed, either
60 // by taking exception straight away or by entering lockup state, below code is not
61 // executed and LSPACT is not cleared.
62 // In case of NoFault or, on successful return from TakePreserveFPException(), the
 current
63 // instruction execution continues and FPCCR.LSPACT will be cleared.
64
65 if isSecure then
66 FPCCR_S.LSPACT = '0';
67 else
68 FPCCR_NS.LSPACT = '0';
69
70 // If the FP state is being treated as Secure then the registers are zeroed
71 if isSecure && FPCCR_S.TS == '1' then
72 for i = 0 to 31
73 S[i] = Zeros(32);
74 FPSCR = Zeros(32);
75 else
76 for i = 0 to 15
77 S[i] = bits(32) UNKNOWN;
78 FPSCR = bits(32) UNKNOWN;
79
80 return;

```

### E2.1.234 ProcessorID

```

1 // ProcessorID
2 // =====
3 // Returns an integer that uniquely identifies the executing PE in the system.
4
5 integer ProcessorID();

```

### E2.1.235 PushCalleeStack

```

1 // PushCalleeStack()
2 // =====
3
4 ExcInfo PushCalleeStack(boolean doTailChain)
5 // allocate space of the correct stack. NOTE: If we are tail chaining we
6 // look at LR instead of CONTROL.SPSEL to work out which stack to use, as
7 // SPSEL can report the wrong stack in tail chaining cases
8 if doTailChain then
9 if LR[3] == '0' then
10 mode = PEmode_Handler;
11 spName = RNameSP_Main_Secure;
12 else
13 mode = PEmode_Thread;
14 spName = if LR[2] == '1' then RNameSP_Process_Secure else RNameSP_Main_Secure;
15 else
16 spName = LookUpSP();
17 mode = CurrentMode();
18
19 // Calculate the address of the base of the callee frame
20 bits(32) frameptr = _SP(spName) - 0x28;
21
22 /* only the stack locations, not the store order, are architected */
23 // Write out integrity signature

```

```

24 integritySig = if HaveFPExt() then 0xFEFA125A[31:1] : LR[4] else 0xFEFA125B[31:0];
25 exc = Stack(frameptr, 0x0, spName, mode, integritySig);
26 // Stack callee registers
27 if exc.fault == NoFault then exc = Stack(frameptr, 0x8, spName, mode, R[4]);
28 if exc.fault == NoFault then exc = Stack(frameptr, 0xC, spName, mode, R[5]);
29 if exc.fault == NoFault then exc = Stack(frameptr, 0x10, spName, mode, R[6]);
30 if exc.fault == NoFault then exc = Stack(frameptr, 0x14, spName, mode, R[7]);
31 if exc.fault == NoFault then exc = Stack(frameptr, 0x18, spName, mode, R[8]);
32 if exc.fault == NoFault then exc = Stack(frameptr, 0x1C, spName, mode, R[9]);
33 if exc.fault == NoFault then exc = Stack(frameptr, 0x20, spName, mode, R[10]);
34 if exc.fault == NoFault then exc = Stack(frameptr, 0x24, spName, mode, R[11]);
35
36 // Update the stack pointer
37 spExc = _SP(spName, TRUE, frameptr);
38 return MergeExcInfo(exc, spExc);

```

### E2.1.236 PushStack

```

1 // PushStack()
2 // =====
3
4 ExcInfo PushStack(boolean secureException, boolean instExecOk)
5 integer framesize;
6 if HaveFPExt() && CONTROL.FPCA == '1' && (IsSecure() || NSACR.CP10 == '1') then
7 if IsSecure() && FPCCR_S.TS == '1' then
8 framesize = 0xA8;
9 else
10 framesize = 0x68;
11 else
12 framesize = 0x20;
13
14 /* allocate space on the correct stack */
15 bits(1) frameptralign;
16 frameptralign = SP[2];
17 frameptr = (SP - framesize) AND NOT(ZeroExtend('100', 32));
18 spName = LookUpSP();
19
20 /* only the stack locations, not the store order, are architected */
21 (retaddr, itstate) = ReturnState(instExecOk);
22 RETPSR_Type retpsr = XPSR[31:0];
23 retpsr.IT = itstate; // see ReturnState() in-line note for information on XPSR.
24 IT bits
25 retpsr.SPREALIGN = frameptralign;
26 retpsr.SFPA = if IsSecure() then CONTROL_S.SFPA else '0';
27
28 mode = CurrentMode();
29 exc = Stack(frameptr, 0x0, spName, mode, R[0]);
30 if exc.fault == NoFault then exc = Stack(frameptr, 0x4, spName, mode, R[1]);
31 if exc.fault == NoFault then exc = Stack(frameptr, 0x8, spName, mode, R[2]);
32 if exc.fault == NoFault then exc = Stack(frameptr, 0xC, spName, mode, R[3]);
33 if exc.fault == NoFault then exc = Stack(frameptr, 0x10, spName, mode, R[12]);
34 if exc.fault == NoFault then exc = Stack(frameptr, 0x14, spName, mode, LR);
35 if exc.fault == NoFault then exc = Stack(frameptr, 0x18, spName, mode, retaddr);
36 if exc.fault == NoFault then exc = Stack(frameptr, 0x1C, spName, mode, retpsr);
37
38 if HaveFPExt() && CONTROL.FPCA == '1' then
39 newExc = DefaultExcInfo();
40 // LSPACT should not be active at the same time as CONTROL.FPCA. This
41 // is a possible attack scenario so raise a SecureFault.
42 lspact = if FPCCR_S.S == '1' then FPCCR_S.LSPACT else FPCCR_NS.LSPACT;
43 if HaveSecurityExt() && lspact == '1' then
44 SFSR.LSERR = '1';
45 newExc = CreateException(SecureFault, TRUE, TRUE);
46 elseif !IsSecure() && NSACR.CP10 == '0' then
47 UFSR_S.NOCP = '1';
48 newExc = CreateException(UsageFault, TRUE, TRUE);
49 else
50 if FPCCR.LSPEN == '0' then

```



Chapter E2. Pseudocode Specification  
E2.1. Alphabetical Pseudocode List

```

50 if exc.fault == NoFault then
51 exc = CheckCPEnabled(10);
52 if exc.fault == NoFault then
53 for i = 0 to 15
54 if exc.fault == NoFault then
55 exc = Stack(frameptr, 0x20+(4*i), spName, mode, S[i]);
56 if exc.fault == NoFault then
57 exc = Stack(frameptr, 0x60, spName, mode, FPSCR);
58 if framesize == 0xA8 then
59 for i = 0 to 15
60 if exc.fault == NoFault then
61 exc = Stack(frameptr, 0x68+(4*i), spName, mode, S[i+16]);
62 (cpEnabled, -) = IsCPEnabled(10);
63 if cpEnabled then
64 if framesize == 0xA8 then
65 for i = 0 to 31
66 S[i] = Zeros(32);
67 FPSCR = Zeros(32);
68 else
69 for i = 0 to 15
70 S[i] = bits(32) UNKNOWN;
71 FPSCR = bits(32) UNKNOWN;
72 else
73 UpdateFPCCR(frameptr + 0x20, TRUE);
74 if newExc.fault != NoFault then
75 // It is IMPLEMENTATION_DEFINED whether to drop the earlier MemFault
76 // if the Secure fault or NOCP fault is also generated subsequently.
77 // If MemFault is not dropped, it will be merged with Secure/NOCP fault
78 // based on exception priority as per MergeExcInfo().
79 if boolean IMPLEMENTATION_DEFINED "Drop previously generated exceptions" then
80 exc = newExc;
81 else
82 exc = MergeExcInfo(exc, newExc);
83
84 // Set the stack pointer to be at the bottom of the new stack frame
85 spExc = _SP(spName, TRUE, frameptr);
86 exc = MergeExcInfo(exc, spExc);
87
88 bit isSecure = if IsSecure() then '1' else '0';
89 bit isThread = if mode == PMode_Thread then '1' else '0';
90 // Some excReturn bits (eg ES, SPSEL) are set by ExceptionTaken
91 if HaveFPExt() then
92 LR = Ones(25):isSecure:'1':NOT(CONTROL.FPCA):isThread:'000';
93 else
94 LR = Ones(25):isSecure:'11':isThread:'000';
95 return exc;

```

E2.1.237 R

```

1 // R[]
2 // ===
3
4 // Non-assignment form
5
6 bits(32) R[integer n]
7 assert n >= 0 && n <= 15;
8 bits(32) result;
9 case n of
10 when 0 result = _R[RName0];
11 when 1 result = _R[RName1];
12 when 2 result = _R[RName2];
13 when 3 result = _R[RName3];
14 when 4 result = _R[RName4];
15 when 5 result = _R[RName5];
16 when 6 result = _R[RName6];
17 when 7 result = _R[RName7];
18 when 8 result = _R[RName8];
19 when 9 result = _R[RName9];

```

```

20 when 10 result = _R[RName10];
21 when 11 result = _R[RName11];
22 when 12 result = _R[RName12];
23 when 13 result = _R[LookUpSP()][31:2]:'00';
24 when 14 result = _R[RName_LR];
25 when 15 result = _R[RName_PC] + 4;
26 return result;
27
28 // Assignment form
29
30 R[integer n] = bits(32) value
31 assert n >= 0 && n <= 14;
32 RName regName;
33 case n of
34 when 0 _R[RName0] = value;
35 when 1 _R[RName1] = value;
36 when 2 _R[RName2] = value;
37 when 3 _R[RName3] = value;
38 when 4 _R[RName4] = value;
39 when 5 _R[RName5] = value;
40 when 6 _R[RName6] = value;
41 when 7 _R[RName7] = value;
42 when 8 _R[RName8] = value;
43 when 9 _R[RName9] = value;
44 when 10 _R[RName10] = value;
45 when 11 _R[RName11] = value;
46 when 12 _R[RName12] = value;
47 when 13
48 // It is IMPLEMENTATION DEFINED whether stack pointer limit checking
49 // is performed for instructions that were previously UNPREDICTABLE
50 // when modifying the stack pointer.
51 if boolean IMPLEMENTATION_DEFINED "SPLim check UNPRED instructions" then
52 = _SP(LookUpSP(), FALSE, value);
53 else
54 _R[LookUpSP()] = value[31:2]:'00';
55 when 14 _R[RName_LR] = value;
56 return;

```

### E2.1.238 RaiseAsyncBusFault

```

1 // RaiseAsyncBusFault()
2 // =====
3
4 RaiseAsyncBusFault()
5 if HaveMainExt() then
6 BFSR.IMPRESISERR = '1';
7
8 excInfo = CreateException(BusFault, FALSE, boolean UNKNOWN, FALSE);
9 HandleException(excInfo);

```

### E2.1.239 RawExecutionPriority

```

1 // RawExecutionPriority()
2 // =====
3 // Determine the current execution priority without the effect of priority boosting
4
5 integer RawExecutionPriority()
6 execPri = HighestPri();
7 for i = 2 to MaxExceptionNum() // IPSR values of the exception handlers
8 for j = 0 to 1 // Check both Non-secure and Secure exceptions
9 secure = (j == 0);
10 if IsActiveForState(i, secure) then
11 // PRIGROUP effect applied in ExceptionPriority
12 effectivePriority = ExceptionPriority(i, secure, TRUE);
13 if effectivePriority < execPri then
14 execPri = effectivePriority;
15 return execPri;

```

**E2.1.240 Replicate**

```

1 // Replicate()
2 // =====
3
4 bits(M*N) Replicate(bits(M) x, integer N);
5
6 bits(N) Replicate(bits(M) x)
7 assert N MOD M == 0;
8 return Replicate(x, N DIV M);

```

**E2.1.241 ResetSCSRegs**

```

1 // ResetSCSRegs
2 // =====
3 // Sets all registers in the System Control Space (SCS) that have
4 // architecturally-defined reset values to those values
5
6 ResetSCSRegs();

```

**E2.1.242 RestrictedNSPri**

```

1 // RestrictedNSPri()
2 // =====
3 // The priority to which Non-secure exceptions are restricted if AIRCR.PRIS is set
4
5 integer RestrictedNSPri()
6 return 0x80;

```

**E2.1.243 ReturnState**

```

1 // ReturnState()
2 // =====
3
4 (bits(32), ITSTATEType) ReturnState(boolean instExecOk)
5
6 // Whether the return address (and associated IT state) point to the current
7 // instruction or the next instruction only depends on whether the
8 // instruction executed correctly, and not the type of exception.
9 //
10 // For trivial cases this behavior matches the following expectation:-
11 // * Faults (eg MemManage, UsageFault, etc) result in the return address
12 // pointing to the instruction that caused the fault.
13 // * Interrupts and SVC's result in the return address pointing to the next
14 // instruction.
15 //
16 // However it is important to realise that the behavior can differ from the
17 // expectation above in complex cases. The following examples illustrate how
18 // and why the behavior can be different:-
19 // 1) A MemManage fault occurring at the same time as a higher priority
20 // interrupt. The interrupt is taken first due to its priority, but the
21 // return address is set to the current instruction because it didn't
22 // execute successfully. This ensures the return state is correct for
23 // when the pending MemManage fault is taken (which may occur by tail
24 // chaining after the interrupt handler returns).
25 // 2) The architecture states:-
26 // "A fault that is escalated to the priority of a HardFault
27 // retains the return address value of the original fault."
28 // So a SVC that escalates to a HardFault has the return address of the
29 // instruction after SVC (because the SVC succeeded is setting an
30 // exception pending).
31 // 3) The BusFault exception is disabled when a BusFault occurs during
32 // lazy FP state preservation. The fault remains pending until a store
33 // instruction re-enables the BusFault by writing to the SHCSR
34 // register, at which point the exception can be taken. However because

```

```

35 // the store instruction didn't cause the fault, it just allowed it to
36 // be taken the return address points to the instruction after the
37 // store.
38 //
39 // NOTE: Asynchronous faults (eg async BusFault) deviate from this rule and
40 // have a return address set to the next instruction. Due to their
41 // asynchronous nature the address of the actual instruction that
42 // caused the fault is not known.
43 //
44 // The return address is always halfword aligned, meaning bit[0] is
45 // always zero. If present the XPSR.IT bits saved to the stack are
46 // consistent with return address.
47 if instExecOk then
48 return (NextInstrAddr(), NextInstrITState());
49 else
50 return (ThisInstrAddr(), ThisInstrITState());

```

### E2.1.244 RName

```

1 // The names of the core registers. SP is a Banked register.
2
3 enumeration RName {RName0, RName1, RName2, RName3, RName4, RName5, RName6,
4 RName7, RName8, RName9, RName10, RName11, RName12,
5 RNameSP_Main_NonSecure, RNameSP_Process_NonSecure, RName_LR, RName_PC,
6 RNameSP_Main_Secure, RNameSP_Process_Secure};

```

### E2.1.245 ROR

```

1 // ROR()
2 // =====
3
4 bits(N) ROR(bits(N) x, integer shift)
5 if shift == 0 then
6 result = x;
7 else
8 (result, -) = ROR_C(x, shift);
9 return result;

```

### E2.1.246 ROR\_C

```

1 // ROR_C()
2 // =====
3
4 (bits(N), bit) ROR_C(bits(N) x, integer shift)
5 assert shift != 0;
6 m = shift MOD N;
7 result = LSR(x,m) OR LSL(x,N-m);
8 carry_out = result[N-1];
9 return (result, carry_out);

```

### E2.1.247 RoundDown

```

1 // RoundDown()
2 // =====
3
4 integer RoundDown(real x);

```

### E2.1.248 RoundTowardsZero

```

1 // RoundTowardsZero()
2 // =====
3
4 integer RoundTowardsZero(real x)
5 return if x == 0.0 then 0 else if x > 0.0 then RoundDown(x) else RoundUp(x);

```

### E2.1.249 RoundUp

```
1 // RoundUp()
2 // =====
3
4 integer RoundUp(real x);
```

### E2.1.250 RRX

```
1 // RRX()
2 // =====
3
4 bits(N) RRX(bits(N) x, bit carry_in)
5 (result, -) = RRX_C(x, carry_in);
6 return result;
```

### E2.1.251 RRX\_C

```
1 // RRX_C()
2 // =====
3
4 (bits(N), bit) RRX_C(bits(N) x, bit carry_in)
5 result = carry_in : x[N-1:1];
6 carry_out = x[0];
7 return (result, carry_out);
```

### E2.1.252 RSPCheck

```
1 // RSPCheck[] - assignment form
2 // =====
3
4 RSPCheck[integer n] = bits(32) value
5 if n == 13 then
6 - = _SP(LookUpSP(), FALSE, value);
7 else
8 R[n] = value;
9 return;
```

### E2.1.253 S

```
1 // S[]
2 // ===
3
4 // Non-assignment form
5
6 bits(32) S[integer n]
7 assert n >= 0 && n <= 31;
8 if (n MOD 2) == 0 then
9 result = D[n DIV 2][31:0];
10 else
11 result = D[n DIV 2][63:32];
12 return result;
13
14 // Assignment form
15
16 S[integer n] = bits(32) value
17 assert n >= 0 && n <= 31;
18 if (n MOD 2) == 0 then
19 D[n DIV 2][31:0] = value;
20 else
21 D[n DIV 2][63:32] = value;
22 return;
```

**E2.1.254 Sat**

```

1 // Sat()
2 // =====
3
4 bits(N) Sat(integer i, integer N, boolean unsigned)
5 result = if unsigned then UnsignedSat(i, N) else SignedSat(i, N);
6 return result;

```

**E2.1.255 SatQ**

```

1 // SatQ()
2 // =====
3
4 (bits(N), boolean) SatQ(integer i, integer N, boolean unsigned)
5 (result, sat) = if unsigned then UnsignedSatQ(i, N) else SignedSatQ(i, N);
6 return (result, sat);

```

**E2.1.256 SAttributes**

```

1 // Security attributes associated with an address
2
3 type SAttributes is (
4 boolean nsc, // Non-secure callability of an address. FALSE = not
5 // callable from the Non-secure state
6 boolean ns, // Security of an address FALSE = Secure, TRUE = Non-secure
7 bits(8) sregion, // The SAU region number
8 boolean srvalid, // Set to 1 if the SAU region number is valid
9 bits(8) iregion, // The IDAU region number
10 boolean irvalid // Set to 1 if the IDAU region number is valid
11)

```

**E2.1.257 SCS\_UpdateStatusRegs**

```

1 // SCS_UpdateStatusRegs()
2 // =====
3 // Update status registers in the System Control Space (SCS)
4
5 SCS_UpdateStatusRegs();

```

**E2.1.258 SecureDebugMonitorAllowed**

```

1 // SecureDebugMonitorAllowed()
2 // =====
3
4 boolean SecureDebugMonitorAllowed()
5 if DAUTHCTRL.SPIDENSEL == '1' then
6 return DAUTHCTRL.INTSPIDEN == '1';
7 else
8 return ExternalSecureSelfHostedDebugEnabled();

```

**E2.1.259 SecureHaltingDebugEnabled**

```

1 // SecureHaltingDebugEnabled()
2 // =====
3
4 boolean SecureHaltingDebugEnabled()
5 if HaltingDebugEnabled() == FALSE then
6 return FALSE;
7 elsif DAUTHCTRL.SPIDENSEL == '1' then
8 return DAUTHCTRL.INTSPIDEN == '1';
9 else
10 return ExternalSecureInvasiveDebugEnabled();

```

## E2.1.260 SecureNoninvasiveDebugAllowed

```
1 // SecureNoninvasiveDebugAllowed()
2 // =====
3
4 boolean SecureNoninvasiveDebugAllowed()
5 if DHCSR.S_SDE == '1' then
6 return TRUE;
7 elseif !NoninvasiveDebugAllowed() then
8 return FALSE;
9 elseif DAUTHCTRL.SPNIDENSEL == '1' then
10 return DAUTHCTRL.INTSPNIDEN == '1';
11 else
12 return ExternalSecureNoninvasiveDebugEnabled();
```

## E2.1.261 SecurityCheck

```
1 // SecurityCheck()
2 // =====
3
4 SAttributes SecurityCheck(bits(32) address, boolean isinstrfetch, boolean isSecure)
5 SAttributes result;
6 addr = UInt(address);
7
8 // Setup default attributes
9 result.ns = FALSE;
10 result.nsc = FALSE;
11 result.sregion = Zeros(8);
12 result.srvalid = FALSE;
13 result.iregion = Zeros(8);
14 result.irvalid = FALSE;
15 idauExempt = FALSE;
16 idauNs = TRUE;
17 idauNsc = TRUE;
18
19 // If an IMPLEMENTATION DEFINED memory security attribution unit is present
20 // query it and override defaults set above. The IDAU is subject to the same
21 // 32byte minimum region granularity as the SAU/MPU.
22 // NOTE: The defaults above are set such that the IDAU has no effect on the
23 // SAU.
24 if boolean IMPLEMENTATION_DEFINED "IDAU present" then
25 (idauExempt,
26 idauNs,
27 idauNsc,
28 result.iregion,
29 result.irvalid) = IDAUCheck(address[31:5]:'00000');
30
31 // The 0xF0000000 -> 0xFFFFFFFF is always Secure for instruction fetches
32 if isinstrfetch && (address[31:28] == '1111') then
33 // Use default attributes defined above
34
35 // Check if the address is exempt from SAU/IDAU checking.
36 elseif idauExempt || // IDAU specified exemption
37 (isinstrfetch && (address[31:28] == '1110')) || // Whole 0xExxxxxxx range
38 ((addr >= 0xE0000000) && (addr <= 0xE0002FFF)) || // ITM, DWT, FPB
39 ((addr >= 0xE000E000) && (addr <= 0xE000EFFF)) || // SCS
40 ((addr >= 0xE002E000) && (addr <= 0xE002EFFF)) || // SCS NS alias
41 ((addr >= 0xE0040000) && (addr <= 0xE0041FFF)) || // TPIU, ETM
42 ((addr >= 0xE00FF000) && (addr <= 0xE00FFFFF)) then // ROM table
43 // memory security reported as NS-Req, and no region information is supplied.
44 result.ns = !isSecure;
45 result.irvalid = FALSE;
46
47 else
48 // If the SAU is enabled check its regions
49 if SAU_CTRL.ENABLE == '1' then
```

```

50 boolean multiRegionHit = FALSE;
51 for r = 0 to (UInt(SAU_TYPE.SREGION) - 1)
52 if SAU_REGION[r].ENABLE == '1' then
53 // SAU region enabled so perform checks
54 bits(32) base_address = SAU_REGION[r].BADDR:'00000';
55 bits(32) limit_address = SAU_REGION[r].LADDR:'11111';
56 if ((UInt(base_address) <= addr) &&
57 (UInt(limit_address) >= addr)) then
58 if result.srvalid then
59 multiRegionHit = TRUE;
60 else
61 result.ns = SAU_REGION[r].NSC == '0';
62 result.nsc = SAU_REGION[r].NSC == '1';
63 result.srvalid = TRUE;
64 result.sregion = r[7:0];
65
66 // If multiple regions are hit then report memory as Secure and not
67 // Non-secure callable. Also don't report any region number
68 // information.
69 if multiRegionHit then
70 result.ns = FALSE;
71 result.nsc = FALSE;
72 result.sregion = Zeros(8);
73 result.srvalid = FALSE;
74
75 // SAU disabled, check if whole address space should be marked as
76 // Non-secure
77 elseif SAU_CTRL.ALLNS == '1' then
78 result.ns = TRUE;
79
80 // Override the internal setting if the external attribution unit
81 // reports more restrictive attributes.
82 if !idauNs then
83 if result.ns || (!idauNsc && result.nsc) then
84 result.ns = FALSE;
85 result.nsc = idauNsc;
86
87 return result;

```

### E2.1.262 SecurityState

```

1 // Type and definition of the current Security state of PE
2
3 enumeration SecurityState {SecurityState_NonSecure, SecurityState_Secure};
4 SecurityState CurrentState;

```

### E2.1.263 SendEvent

```

1 // SendEvent
2 // =====
3 // Performs a send event by setting the Event Register of every PE in multiprocessor system
4
5 SendEvent();

```

### E2.1.264 SerializeVFP

```

1 // SerializeVFP
2 // =====
3 // Ensures that any exceptional conditions in previous floating-point
4 // instructions have been detected
5
6 SerializeVFP();

```

### E2.1.265 SetActive



```

1 // SetActive()
2 // =====
3
4 SetActive(integer exception, boolean isSecure, boolean setNotClear)
5 if !HaveSecurityExt() then
6 isSecure = FALSE;
7 // If the exception target state is configurable there is only one active
8 // bit. To represent this the Non-secure and Secure instances of the active
9 // flags in the array are always set to the same value.
10 if IsExceptionTargetConfigurable(exception) then
11 if ExceptionTargetsSecure(exception, boolean UNKNOWN) == isSecure then
12 ExceptionActive[exception] = if setNotClear then '11' else '00';
13 else
14 idx = if isSecure then 0 else 1;
15 ExceptionActive[exception][idx] = if setNotClear then '1' else '0';

```

### E2.1.266 SetDWTDebugEvent

```

1 // SetDWTDebugEvent()
2 // =====
3 // Set a pending debug event to the PE
4
5 boolean SetDWTDebugEvent(boolean secure_match)
6 if CanHaltOnEvent(secure_match) then
7 DHCSR.C_HALT = '1';
8 DFSR.DWTTRAP = '1';
9 return TRUE;
10
11 elsif HaveMainExt() && CanPendMonitorOnEvent(secure_match, TRUE) then
12 DEMCR.MON_PEND = '1';
13 DFSR.DWTTRAP = '1';
14 return TRUE;
15
16 else
17 return FALSE;

```

### E2.1.267 SetEventRegister

```

1 // SetEventRegister()
2 // =====
3 // Set the Event Register of the current PE
4
5 SetEventRegister();

```

### E2.1.268 SetExclusiveMonitors

```

1 // SetExclusiveMonitors()
2 // =====
3
4 SetExclusiveMonitors(bits(32) address, integer size)
5
6 boolean isSecure = CurrentState == SecurityState_Secure;
7 if address != Align(address, size) then
8 UFSR.UNALIGNED = '1';
9 excInfo = CreateException(UsageFault, FALSE, isSecure);
10 else
11 (excInfo, memaddrdesc) = ValidateAddress(address, AccType_NORMAL, FindPriv(),
12 isSecure, FALSE, TRUE);
13 HandleException(excInfo);
14
15 if memaddrdesc.memattrs.shareable then
16 MarkExclusiveGlobal(memaddrdesc.paddress, ProcessorID(), size);
17
18 MarkExclusiveLocal(memaddrdesc.paddress, ProcessorID(), size);

```

**E2.1.269 SetITSTATEAndCommit**

```

1 // SetITSTATEAndCommit()
2 // =====
3
4 SetITSTATEAndCommit(ITSTATEType it)
5 // This function directly commits the change to the ITSTATE, so ThisInstrITSTATE()
6 // and NextInstrITSTATE() both point to the target address.
7 _NextInstrITState = it;
8 _ITStateChanged = TRUE;
9 EPSR.IT = it;
10 return;

```

**E2.1.270 SetMonStep**

```

1 // SetMonStep()
2 // =====
3 // Check whether DebugMonitor priority is still sufficient for debug stepping after the
4 // execution of current instruction. If monitor step is enabled before the
5 // execution of the instruction and the priority remains sufficient after the execution
6 // of current instruction, then MON_PEND bit is set for current instruction.
7
8 SetMonStep(boolean mon_step_active)
9
10 // Check whether Monitor Step is enabled at the start of the instruction
11 if !mon_step_active then return;
12
13 // if Monitor Step is enabled, check whether current instruction has cleared MON_STEP bit
14 if DEMCR.MON_STEP == '0' then UNPREDICTABLE;
15
16 // Check whether DebugMonitor priority remains greater-than the current priority, and if
17 // so,
18 // set the MON_PEND bit.
19 if ExceptionPriority(DebugMonitor, IsSecure(), TRUE) < ExecutionPriority() then
20 DEMCR.MON_PEND = '1';
21 DFSR.HALTED = '1';
22 return;

```

**E2.1.271 SetPending**

```

1 // SetPending()
2 // =====
3
4 SetPending(integer exception, boolean isSecure, boolean setNotClear)
5 if !HaveSecurityExt() then
6 isSecure = FALSE;
7 // If the exception target state is configurable there is only one pending
8 // bit. To represent this, the Non-secure and Secure instances of the pending
9 // flags in the array are always set to the same value.
10 if IsExceptionTargetConfigurable(exception) then
11 ExceptionPending[exception] = if setNotClear then '11' else '00';
12 else
13 idx = if isSecure then 0 else 1;
14 ExceptionPending[exception][idx] = if setNotClear then '1' else '0';

```

**E2.1.272 SetThisInstrDetails**

```

1 // SetThisInstrDetails
2 // =====
3 // Set the details of current instruction
4
5 SetThisInstrDetails(bits(32) opcode, integer len, bits(4) defaultCond);

```

**E2.1.273 Shift**

```

1 // Shift()
2 // =====
3
4 bits(N) Shift(bits(N) value, SType sr_type, integer amount, bit carry_in)
5 (result, -) = Shift_C(value, sr_type, amount, carry_in);
6 return result;

```

### E2.1.274 Shift\_C

```

1 // Shift_C()
2 // =====
3
4 (bits(N), bit) Shift_C(bits(N) value, SType sr_type, integer amount, bit carry_in)
5 assert !(sr_type == SType_RRX && amount != 1);
6
7 if amount == 0 then
8 (result, carry_out) = (value, carry_in);
9 else
10 case sr_type of
11 when SType_LSL
12 (result, carry_out) = LSL_C(value, amount);
13 when SType_LSR
14 (result, carry_out) = LSR_C(value, amount);
15 when SType_ASR
16 (result, carry_out) = ASR_C(value, amount);
17 when SType_ROR
18 (result, carry_out) = ROR_C(value, amount);
19 when SType_RRX
20 (result, carry_out) = RRX_C(value, carry_in);
21
22 return (result, carry_out);

```

### E2.1.275 SignedSat

```

1 // SignedSat()
2 // =====
3
4 bits(N) SignedSat(integer i, integer N)
5 (result, -) = SignedSatQ(i, N);
6 return result;

```

### E2.1.276 SignedSatQ

```

1 // SignedSatQ()
2 // =====
3
4 (bits(N), boolean) SignedSatQ(integer i, integer N)
5 if i > 2^(N-1) - 1 then
6 result = 2^(N-1) - 1; saturated = TRUE;
7 elsif i < -(2^(N-1)) then
8 result = -(2^(N-1)); saturated = TRUE;
9 else
10 result = i; saturated = FALSE;
11 return (result[N-1:0], saturated);

```

### E2.1.277 SignExtend

```

1 // SignExtend()
2 // =====
3
4 bits(N) SignExtend(bits(M) x, integer N)
5 assert N >= M;
6 return Replicate(x[M-1], N-M) : x;
7

```

```

8 bits(N) SignExtend(bits(M) x)
9 return SignExtend(x, N);

```

### E2.1.278 SleepOnExit

```

1 // SleepOnExit()
2 // =====
3 // Optionally returns PE to a power-saving mode on return from the only
4 // active exception
5
6 SleepOnExit();

```

### E2.1.279 SP

```

1 // SP
2 // ==
3
4 // Non-assignment form
5
6 bits(32) SP
7 return R[13];
8
9 // Assignment form
10
11 SP = bits(32) value
12 RSPCheck[13] = value;

```

### E2.1.280 SP\_Main

```

1 // SP_Main
2 // =====
3
4 // Non-assignment form
5
6 bits(32) SP_Main
7 value = if IsSecure() then SP_Main_Secure else SP_Main_NonSecure;
8 return value;
9
10 // Assignment form
11
12 SP_Main = bits(32) value
13 if IsSecure() then
14 SP_Main_Secure = value;
15 else
16 SP_Main_NonSecure = value;

```

### E2.1.281 SP\_Main\_NonSecure

```

1 // SP_Main_NonSecure
2 // =====
3
4 // Non-assignment form
5
6 bits(32) SP_Main_NonSecure
7 return _SP(RNameSP_Main_NonSecure);
8
9 // Assignment form
10
11 SP_Main_NonSecure = bits(32) value
12 - = _SP(RNameSP_Main_NonSecure, FALSE, value);

```

### E2.1.282 SP\_Main\_Secure

```

1 // SP_Main_Secure
2 // =====
3
4 // Non-assignment form
5
6 bits(32) SP_Main_Secure
7 return _SP(RNameSP_Main_Secure);
8
9 // Assignment form
10
11 SP_Main_Secure = bits(32) value
12 - = _SP(RNameSP_Main_Secure, FALSE, value);

```

**E2.1.283 SP\_Process**

```

1 // SP_Process
2 // =====
3
4 // Non-assignment form
5
6 bits(32) SP_Process
7 value = if IsSecure()
8 then SP_Process_Secure else SP_Process_NonSecure;
9 return value;
10
11 // Assignment form
12
13 SP_Process = bits(32) value
14 if IsSecure() then
15 SP_Process_Secure = value;
16 else
17 SP_Process_NonSecure = value;

```

**E2.1.284 SP\_Process\_NonSecure**

```

1 // SP_Process_NonSecure
2 // =====
3
4 // Non-assignment form
5
6 bits(32) SP_Process_NonSecure
7 return _SP(RNameSP_Process_NonSecure);
8
9 // Assignment form
10
11 SP_Process_NonSecure = bits(32) value
12 - = _SP(RNameSP_Process_NonSecure, FALSE, value);

```

**E2.1.285 SP\_Process\_Secure**

```

1 // SP_Process_Secure
2 // =====
3
4 // Non-assignment form
5
6 bits(32) SP_Process_Secure
7 return _SP(RNameSP_Process_Secure);
8
9 // Assignment form
10
11 SP_Process_Secure = bits(32) value
12 - = _SP(RNameSP_Process_Secure, FALSE, value);

```

**E2.1.286 SpeculativeSynchronizationBarrier**

```

1 // Speculative Synchronisation Barrier
2 // =====
3 // Perform a Speculative Synchronization Barrier
4
5 SpeculativeSynchronizationBarrier();

```

### E2.1.287 SRTYPE

```

1 // Different types of shift and rotate operations
2 enumeration SRTYPE {SRTYPE_LSL, SRTYPE_LSR, SRTYPE_ASR, SRTYPE_ROR, SRTYPE_RRX};

```

### E2.1.288 Stack

```

1 // Stack
2 // =====
3
4 // Assignment form
5
6 ExcInfo Stack(bits(32) frameptr, integer offset, RName spreg, PMode mode, bits(32) value)
7 // This function is used to perform register stacking operations that are
8 // done around exception handling. If the stack pointer is below the stack
9 // pointer limit but the access itself is above the limit it is
10 // IMPLEMENTATION DEFINED whether the write is performed. If the
11 // address of access is below the limit the access is not performed
12 // regardless of the stack pointer value.
13 (limit, applylimit) = LookUpSPLim(spreg);
14 if !applylimit || (UInt(frameptr) >= UInt(limit)) then
15 doAccess = TRUE;
16 else
17 doAccess = boolean IMPLEMENTATION_DEFINED "Push non-violating locations";
18
19 address = frameptr + offset;
20 if doAccess && (!applylimit || ((UInt(address) >= UInt(limit)))) then
21 secure = ((spreg == RNameSP_Main_Secure) ||
22 (spreg == RNameSP_Process_Secure));
23 // Work out if the stack operations should be privileged or not
24 if secure then
25 isPriv = CONTROL_S.nPRIV == '0';
26 else
27 isPriv = CONTROL_NS.nPRIV == '0';
28 isPriv = isPriv || (mode == PMode_Handler);
29 // Finally perform the memory operations
30 excInfo = MemA_with_priv_security(address, 4, AccType_STACK, isPriv, secure, TRUE, value);
31 else
32 excInfo = DefaultExcInfo();
33 return excInfo;
34
35 // Non-assignment form
36
37 (ExcInfo, bits(32)) Stack(bits(32) frameptr, integer offset, RName spreg, PMode mode)
38 secure = ((spreg == RNameSP_Main_Secure) ||
39 (spreg == RNameSP_Process_Secure));
40 // Work out if the stack operations should be privileged or not
41 if secure then
42 isPriv = CONTROL_S.nPRIV == '0';
43 else
44 isPriv = CONTROL_NS.nPRIV == '0';
45 isPriv = isPriv || (mode == PMode_Handler);
46 // Finally perform the memory operations
47 address = frameptr + offset;
48 (excInfo, value) = MemA_with_priv_security(address, 4, AccType_STACK, isPriv, secure, TRUE);
49 return (excInfo, value);

```

### E2.1.289 StandardFPSCRValue

```

1 // StandardFPSCRValue()
2 // =====
3
4 bits(32) StandardFPSCRValue()
5 return '00000' : FPSCR[26] : '11000000000000000000000000000000';

```

### E2.1.290 SteppingDebug

```

1 // SteppingDebug()
2 // =====
3 // At the start of each instruction execution, check for debug stepping.
4 // This function does not cover the scenario where the instruction being stepped raises
5 // another
6 // exception, or returns from an exception and enters/tailchains into another exception
7 // without
8 // executing the instruction in background code.
9
10 boolean SteppingDebug()
11 // If Halting debug is allowed and C_STEP is set, set C_HALT for the next instruction.
12 if CanHaltOnEvent(IsSecure()) && DHCSR.C_STEP == '1' then
13 DHCSR.C_HALT = '1';
14 DFSR.HALTED = '1';
15
16 // If the current execution priority is below DebugMonitor and generating a DebugMonitor
17 // exception is allowed, and MON_STEP is set, then return TRUE. Otherwise return FALSE.
18 // This is used to determine whether to set MON_PEND for the next instruction if the
19 // execution priority remains below DebugMonitor.
20 mon_step_enabled = HaveDebugMonitor() && CanPendMonitorOnEvent(IsSecure(), TRUE);
21 return (mon_step_enabled && DEMCR.MON_STEP == '1');

```

### E2.1.291 T32ExpandImm

```

1 // T32ExpandImm()
2 // =====
3
4 bits(32) T32ExpandImm(bits(12) imm12)
5
6 // APSR.C argument to following function call does not affect the imm32 result.
7 (imm32, -) = T32ExpandImm_C(imm12, APSR.C);
8
9 return imm32;

```

### E2.1.292 T32ExpandImm\_C

```

1 // T32ExpandImm_C()
2 // =====
3
4 (bits(32), bit) T32ExpandImm_C(bits(12) imm12, bit carry_in)
5
6 if imm12[11:10] == '00' then
7
8 case imm12[9:8] of
9 when '00'
10 imm32 = ZeroExtend(imm12[7:0], 32);
11 when '01'
12 if imm12[7:0] == '00000000' then UNPREDICTABLE;
13 imm32 = '00000000' : imm12[7:0] : '00000000' : imm12[7:0];
14 when '10'
15 if imm12[7:0] == '00000000' then UNPREDICTABLE;
16 imm32 = imm12[7:0] : '00000000' : imm12[7:0] : '00000000';
17 when '11'
18 if imm12[7:0] == '00000000' then UNPREDICTABLE;
19 imm32 = imm12[7:0] : imm12[7:0] : imm12[7:0] : imm12[7:0];
20 carry_out = carry_in;
21

```

```

22 else
23
24 unrotated_value = ZeroExtend('1':imm12[6:0], 32);
25 (imm32, carry_out) = ROR_C(unrotated_value, UInt(imm12[11:7]));
26
27 return (imm32, carry_out);

```

### E2.1.293 TailChain

```

1 // TailChain()
2 // =====
3
4 ExcInfo TailChain(integer exceptionNumber, boolean excIsSecure, EXC_RETURN_Type excReturn)
5 // Refresh LR with the excReturn value, ready for the next exception
6 if !HaveFPEExt() then
7 excReturn.FType = '1';
8 excReturn.PREFIX = Ones(8);
9 LR = excReturn;
10
11 return ExceptionTaken(exceptionNumber, TRUE, excIsSecure, FALSE);

```

### E2.1.294 TakePreserveFPEException

```

1 // TakePreserveFPEException()
2 // =====
3
4 TakePreserveFPEException(ExcInfo excInfo)
5 assert HaveFPEExt();
6 assert excInfo.origFault IN {DebugMonitor, SecureFault, MemManage, BusFault, UsageFault};
7
8 // Get the details of the original fault so that any escalation to HardFault / Lockup
9 // based
10 // on the current execution priority is ignored. Escalation is performed manually against
11 // the FPCCR.*RDP fields below.
12 exception = excInfo.origFault;
13 isSecure = excInfo.origFaultIsSecure;
14 fpccr = if isSecure then FPCCR_S else FPCCR_NS;
15
16 if FPCCR_S.MONRDY == '1' && FPCCR_S.HFRDY == '0' then UNPREDICTABLE;
17 if FPCCR_S.BFRDY == '1' && FPCCR_S.HFRDY == '0' then UNPREDICTABLE;
18 if FPCCR_S.SFRDY == '1' && FPCCR_S.HFRDY == '0' then UNPREDICTABLE;
19 if fpccr.UFRDY == '1' && FPCCR_S.HFRDY == '0' then UNPREDICTABLE;
20 if fpccr.MMRDY == '1' && FPCCR_S.HFRDY == '0' then UNPREDICTABLE;
21 if exception == DebugMonitor && FPCCR_S.MONRDY == '0' then
22 // ignore DebugMonitor exception
23 return;
24
25 // Handle exception specific details like escalation and syndrome information
26 case exception of
27 when MemManage
28 escalate = fpccr.MMRDY == '0';
29 when UsageFault
30 escalate = fpccr.UFRDY == '0';
31 when BusFault
32 escalate = FPCCR_S.BFRDY == '0';
33 when SecureFault
34 escalate = FPCCR_S.SFRDY == '0';
35 otherwise
36 escalate = FALSE;
37 if escalate then
38 exception = HardFault;
39 // Faults that originally targeted the Secure state still target the
40 // Secure state even if HardFault normally targets Non-secure.
41 isSecure = isSecure || ExceptionTargetsSecure(HardFault, isSecure);
42
43 // Check if the exception is enabled and has sufficient priority to
44 // preempt and be taken straight away.

```



```

44 if (ExceptionPriority(exception, isSecure, TRUE) < ExecutionPriority()) &&
45 ExceptionEnabled(exception, isSecure) then
46 if escalate then
47 HFSR.FORCED = '1';
48 // Set the exception pending and terminate the current instruction. This
49 // leaves FP disabled (that is CONTROL.FPCA set to 0) and prevents the
50 // preempting exception entry reserving space for a redundant FP state.
51 SetPending(exception, isSecure, TRUE);
52 EndOfInstruction();
53 else
54 // If the reason the exception cannot preempt is because of the fact that
55 // HardFault couldn't be entered by the context the FP state belongs to
56 // then enter the lockup state.
57 if FPCCR_S.HFRDY == '0' then
58 Lockup(TRUE); // Lockup at current priority, lock-up address = 0xEFFFFFFE
59 else
60 if escalate then
61 HFSR.FORCED = '1';
62 // Set the exception pending so it will be taken after the current
63 // handler returns.
64 SetPending(exception, isSecure, TRUE);
65 return;

```

### E2.1.295 TakeReset

```

1 // TakeReset ()
2 // =====
3
4 TakeReset ()
5 // If the Security Extension is implemented the PE resets into Secure state.
6 // If the Security Extension is not implemented the PE resets into Non-secure state.
7 CurrentState = if HaveSecurityExt () then SecurityState_Secure else
8 SecurityState_NonSecure;
9
10 ResetSCSRegs (); // Catch-all function for System Control Space reset
11 APSR = bits(32) UNKNOWN; // Flags UNPREDICTABLE from reset
12 IPSR.Exception = Zeros(9); // Exception number cleared at reset
13 if HaveMainExt () then
14 LR = Ones(32); // Preset to an illegal exception return value
15 SetITSTATEAndCommit(Zeros(8)); // IT/ICI bits cleared
16 else
17 LR = bits(32) UNKNOWN; // Value must be initialised by software
18
19 // Reset priority boosting
20 PRIMASK_NS[0] = '0'; // priority mask cleared at reset
21 if HaveSecurityExt () then
22 PRIMASK_S[0] = '0';
23 if HaveMainExt () then
24 FAULTMASK_NS[0] = '0'; // Fault mask cleared at reset
25 BASEPRI_NS[7:0] = Zeros(8); // Base priority disabled at reset
26 if HaveSecurityExt () then
27 FAULTMASK_S[0] = '0';
28 BASEPRI_S[7:0] = Zeros(8);
29
30 // Initialize the Floating Point Extn
31 if HaveFPExt () then
32 CONTROL.FPCA = '0'; // FP inactive
33 FPDSCR_NS.AHP = '0';
34 FPDSCR_NS.DN = '0';
35 FPDSCR_NS.FZ = '0';
36 FPDSCR_NS.RMode = '00';
37 FPCCR.LSPEN = '1';
38 FPCCR_NS.ASPEN = '1';
39 FPCCR_NS.LSPACT = '0';
40 FPCAR_NS = bits(32) UNKNOWN;
41 if HaveSecurityExt () then
42 CONTROL_S.SFPA = '0';
43 FPDSCR_S.AHP = '0';

```

```

43 FPDSCR_S.DN = '0';
44 FPDSCR_S.FZ = '0';
45 FPDSCR_S.RMode = '00';
46 FPCCR.LSPENS = '0';
47 FPCCR_S.ASPEN = '1';
48 FPCCR_S.LSPACT = '0';
49 FPCAR_S = bits(32) UNKNOWN;
50 for i = 0 to 31
51 S[i] = bits(32) UNKNOWN;
52
53 // Thread is privileged, current stack is Main
54 CONTROL_NS.SPSEL = '0';
55 CONTROL_NS.nPRIV = '0';
56 if HaveSecurityExt() then
57 CONTROL_S.SPSEL = '0';
58 CONTROL_S.nPRIV = '0';
59
60 for i = 0 to MaxExceptionNum() // All exceptions Inactive
61 ExceptionActive[i] = '00';
62 ClearExclusiveLocal(ProcessorID()); // Synchronization (LDREX* / STREX*) monitor
63 ClearEventRegister(); // See WFE instruction for more information
64 for i = 0 to 12
65 R[i] = bits(32) UNKNOWN;
66
67 // Clearing stack limit registers
68 if HaveMainExt() then
69 MSPLIM_NS = Zeros(32);
70 PSPLIM_NS = Zeros(32);
71 if HaveSecurityExt() then
72 MSPLIM_S = Zeros(32);
73 PSPLIM_S = Zeros(32);
74
75 // Load the initial value of the stack pointer and the reset value from the
76 // vector table. The order of the loads is IMPLEMENTATION DEFINED
77 (excSp, sp) = Vector[0, HaveSecurityExt()];
78 (excRst, start) = Vector[Reset, HaveSecurityExt()];
79 if excSp.fault != NoFault || excRst.fault != NoFault then
80 Lockup(TRUE);
81
82 // Initialize the stack pointers and start execution at the reset vector
83 if HaveSecurityExt() then
84 SP_Main_Secure = sp;
85 SP_Main_NonSecure = ((bits(30) UNKNOWN):'00');
86 SP_Process_Secure = ((bits(30) UNKNOWN):'00');
87 else
88 SP_Main_NonSecure = sp;
89 SP_Process_NonSecure = ((bits(30) UNKNOWN):'00');
90 EPSR.T = start[0];
91 BranchToAndCommit(start[31:1]:'0');

```

### E2.1.296 ThisInstr

```

1 // ThisInstr
2 // =====
3 // Returns a 32-bit value which contain the bitstring encoding of current instruction.
4 // In case of 16-bit instructions, the instruction is packed into the bottom 16-bits
5 // with upper 16-bits zeroed. In case of 32-bit instructions, the instruction is
6 // treated as two halfwords, with the first halfword of the instruction in the
7 // top 16-bits and second halfword in bottom 16-bits.
8
9 bits(32) ThisInstr();

```

### E2.1.297 ThisInstrAddr

```

1 // ThisInstrAddr()
2 // =====

```

```

3
4 bits(32) ThisInstrAddr()
5 return _R[RName_PC];

```

### E2.1.298 ThisInstrITState

```

1 // ThisInstrITState()
2 // =====
3
4 ITSTATEType ThisInstrITState()
5 if HaveMainExt() then
6 value = EPSR.IT;
7 else
8 value = Zeros(8);
9 return value;

```

### E2.1.299 ThisInstrLength

```

1 // ThisInstrLength
2 // =====
3 // Returns the length of the current instruction in bytes
4
5 integer ThisInstrLength();

```

### E2.1.300 TopLevel

```

1 // TopLevel()
2 // =====
3
4 // This function is called one time for each tick the PE is not in a sleep
5 // state. It handles all instruction processing, including fetching the opcode,
6 // decode and execute. It also handles pausing execution when in the lockup
7 // state.
8 TopLevel()
9 // If the PE is locked up then abort execution of this instruction. Set
10 // the length of the current instruction to 0 so NextInstrAddr() reports the
11 // correct lockup address.
12 ok = DHCSR.S_LOCKUP != '1';
13 if !ok then
14 SetThisInstrDetails(Zeros(32), 0, Ones(4));
15 else
16 // Check for stepping debug for current instruction fetch.
17 mon_step_active = SteppingDebug();
18 UpdateSecureDebugEnabled();
19 pc = ThisInstrAddr();
20
21 try
22 // Not locked up, so attempt to fetch the instruction
23 (instr, is16bit) = FetchInstr(pc);
24
25 // Setup the details of the instruction. NOTE: The default condition
26 // is based on the ITSTATE, however this is overridden in the decode
27 // stage by instructions that have explicit condition codes.
28 len = if is16bit then 2 else 4;
29
30 defaultCond = if ITSTATE[3:0] == '0000' then 0xE[3:0] else ITSTATE[7:4];
31 SetThisInstrDetails(instr, len, defaultCond);
32
33 // Checking for FPB Breakpoint on instructions
34 if HaveFPB() && FPB_CheckBreakPoint(pc, len, TRUE, IsSecure()) then
35 FPB_BreakpointMatch();
36
37 // Finally try and execute the instruction
38 DecodeExecute(instr, pc, is16bit);
39

```

```

40 // Check for Monitor Step
41 if HaveDebugMonitor() then SetMonStep(mon_step_active);
42
43 // Check for DWT match
44 if IsDWTEabled() then DWT_InstructionMatch(pc);
45
46 catch exn
47 when IsSEE(exn) || IsUNDEFINED(exn)
48 // Unallocated instructions in the NOP hint space and instructions
49 // that fail their condition tests are treated like NOP's.
50 nopHint = instr IN {'0000000000000001011111xxxx0000',
51 '111100111010111110000000xxxxxxx'};
52 if ConditionHolds(CurrentCond()) && !nopHint then
53 ok = FALSE;
54 toSecure = IsSecure();
55 // Unallocated instructions in the coprocessor space behave as NOCP
56 // if the coprocessor is disabled.
57 (isCp, cpNum) = IsCPInstruction(instr);
58 if isCp then
59 (cpEnabled, cpFaultState) = IsCPEnabled(cpNum);
60 if isCp && !cpEnabled then
61 // A PE is permitted to decode the coprocessor space and raise
62 // UNDEFINSTR UsageFaults for unallocated encodings even if the
63 // coprocessor is disabled.
64 if boolean IMPLEMENTATION_DEFINED "Decode CP space" then
65 UFSR.UNDEFINSTR = '1';
66 else
67 UFSR.NOCP = '1';
68 toSecure = cpFaultState;
69 else
70 UFSR.UNDEFINSTR = '1';
71
72 // If Main Extension is not implemented the fault will escalate
73 // to a HardFault.
74 excInfo = CreateException(UsageFault, TRUE, toSecure);
75 // Prevent EndOfInstruction() being called in
76 // HandleException() as the instruction has already been
77 // terminated so there is no need to throw the exception
78 // again.
79 excInfo.termInst = FALSE;
80 HandleException(excInfo);
81 when IsExceptionTaken(exn)
82 ok = FALSE;
83 // Do not catch UNPREDICTABLE or internal errors
84
85 // If there is a reset pending do that, otherwise process the normal
86 // instruction advance.
87 try
88 if ExceptionPending[Reset] != '00' then
89 ExceptionPending[Reset] = '00';
90 TakeReset();
91 else
92 // Call instruction advance for exception handling and PC/ITSTATE
93 // advance.
94 InstructionAdvance(ok);
95 catch exn
96 // Do not catch UNPREDICTABLE or internal errors
97 when IsExceptionTaken(exn)
98 // The correct architectural behavior for any exceptions is
99 // performed inside TakeReset() and InstructionAdvance(). So no
100 // additional actions are required in this catch block.

```

### E2.1.301 TTResp

```

1 // TTResp()
2 // =====
3
4 bits(32) TTResp(bits(32) address, boolean alt, boolean forceunpriv)

```

```

5 TT_RESP_Type resp = Zeros();
6
7 // Only allow security checks if currently in Secure state
8 if IsSecure() then
9 sAttributes = SecurityCheck(address, FALSE, IsSecure());
10 if sAttributes.srvalid then
11 resp.SREGION = sAttributes.sregion;
12 resp.SRVALID = '1';
13 if sAttributes.irvalid then
14 resp.IREGION = sAttributes.iregion;
15 resp.IRVALID = '1';
16 addrSecure = if sAttributes.ns then '0' else '1';
17 resp.S = addrSecure;
18
19 // MPU region information only available when privileged or when
20 // inspecting the other MPU state.
21 other_domain = (alt != IsSecure());
22 if CurrentModeIsPrivileged() || alt then
23 (write, read, region, hit) = IsAccessible(address, forceunpriv, other_domain);
24 if hit then
25 resp.MREGION = region;
26 resp.MRVALID = '1';
27 resp.R = read;
28 resp.RW = write;
29 if IsSecure() then
30 resp.NSR = read AND NOT addrSecure;
31 resp.NSRW = write AND NOT addrSecure;
32
33 return resp;

```

### E2.1.302 UnsignedSat

```

1 // UnsignedSat()
2 // =====
3
4 bits(N) UnsignedSat(integer i, integer N)
5 (result, -) = UnsignedSatQ(i, N);
6 return result;

```

### E2.1.303 UnsignedSatQ

```

1 // UnsignedSatQ()
2 // =====
3
4 (bits(N), boolean) UnsignedSatQ(integer i, integer N)
5 if i > 2^N - 1 then
6 result = 2^N - 1; saturated = TRUE;
7 elsif i < 0 then
8 result = 0; saturated = TRUE;
9 else
10 result = i; saturated = FALSE;
11 return (result[N-1:0], saturated);

```

### E2.1.304 UpdateFPCCR

```

1 // UpdateFPCCR()
2 // =====
3
4 UpdateFPCCR(bits(32) frameptr, boolean applySpLim)
5 assert(HaveFPExt());
6
7 FPCAR.ADDRESS = frameptr[31:3];
8 // Flag if the context address violates the stack pointer limit. If the
9 // limit has been violated PreserveFPState() will zero the registers if
10 // required, but will not save the context to the stack.

```

```

11 (limit, limitValid) = LookUpSPLim(LookUpSP());
12 if applySpLim && limitValid && (UInt(frameptr) < UInt(limit)) then
13 FPCCR.SPLIMVIOL = '1';
14 else
15 FPCCR.SPLIMVIOL = '0';
16 FPCCR.LSPACT = '1';
17
18 execPri = ExecutionPriority();
19 isSecure = IsSecure();
20 FPCCR_S.S = if isSecure then '1' else '0';
21 if CurrentModeIsPrivileged() then
22 FPCCR.USER = '0';
23 else
24 FPCCR.USER = '1';
25 if CurrentMode() == PEmode_Thread then
26 FPCCR.THREAD = '1';
27 else
28 FPCCR.THREAD = '0';
29 if execPri > -1 then
30 FPCCR_S.HFRDY = '1';
31 else
32 FPCCR_S.HFRDY = '0';
33 targetSecure = AIRCR.BFHFNMINS == '0';
34 busfaultpri = ExceptionPriority(BusFault, targetSecure, FALSE);
35 if SHCSR_S.BUSFAULTENA == '1' && execPri > busfaultpri then
36 FPCCR_S.BFRDY = '1';
37 else
38 FPCCR_S.BFRDY = '0';
39 memfaultpri = ExceptionPriority(MemManage, isSecure, FALSE);
40 if SHCSR_S.MEMFAULTENA == '1' && execPri > memfaultpri then
41 FPCCR_S.MMRDY = '1';
42 else
43 FPCCR_S.MMRDY = '0';
44 usagefaultpri = ExceptionPriority(UsageFault, FALSE, FALSE);
45 if SHCSR_NS.USGFAULTENA == '1' && execPri > usagefaultpri then
46 FPCCR_NS.UFRDY = '1';
47 else
48 FPCCR_NS.UFRDY = '0';
49 usagefaultpri = ExceptionPriority(UsageFault, TRUE, FALSE);
50 if SHCSR_S.USGFAULTENA == '1' && execPri > usagefaultpri then
51 FPCCR_S.UFRDY = '1';
52 else
53 FPCCR_S.UFRDY = '0';
54 if HaveSecurityExt() then
55 securefaultpri = ExceptionPriority(SecureFault, TRUE, FALSE);
56 if SHCSR_S.SECUREFAULTENA == '1' && execPri > securefaultpri then
57 FPCCR_S.SFRDY = '1';
58 else
59 FPCCR_S.SFRDY = '0';
60 monpri = ExceptionPriority(DebugMonitor, DEMCR.SDME == '1', FALSE);
61 if DEMCR.MON_EN == '1' && execPri > monpri then
62 FPCCR_S.MONRDY = '1';
63 else
64 FPCCR_S.MONRDY = '0';
65 return;

```

### E2.1.305 UpdateSecureDebugEnabled

```

1 // UpdateSecureDebugEnabled()
2 // =====
3 // Update DHCSR.S_SDE and DEMCR.SDME for each instruction
4
5 UpdateSecureDebugEnabled()
6
7 // DHCSR.S_SDE is frozen if the PE is in Debug state
8 if DHCSR.S_HALT == '0' then
9 DHCSR.S_SDE = (if SecureHaltingDebugEnabled() then '1' else '0');
10

```

```

11 // DEMCR.SDME is frozen if DebugMonitor is active or pending
12 if HaveDebugMonitor() && ExceptionActive[DebugMonitor] == '00' && DEMCR.MON_PEND == '0'
13 then
14 DEMCR.SDME = (if SecureDebugMonitorAllowed() then '1' else '0');

```

### E2.1.306 ValidateAddress

```

1 // ValidateAddress()
2 // =====
3
4 (ExcInfo, AddressDescriptor) ValidateAddress(bits(32) address, AccType acctype,
5 boolean ispriv, boolean secure,
6 boolean iswrite, boolean aligned)
7
8 AddressDescriptor result;
9 Permissions perms;
10 ns = boolean UNKNOWN;
11 excInfo = DefaultExcInfo();
12 isInstrfetch = acctype == AccType_IFETCH;
13
14 // Security checking and MPU bank selection if Security Extensions are present.
15 if HaveSecurityExt() then
16 // Check SAU/IDAU for given address.
17 sAttrib = SecurityCheck(address, isInstrfetch, secure);
18 if isInstrfetch then
19 ns = sAttrib.ns;
20 secureMpu = !sAttrib.ns;
21 // Override the privilege flag supplied with the a value based on the
22 // privilege associated with the current mode and the Security state
23 // of the MPU being queried. This can be different from value this
24 // function is called with, because CONTROL.nPRIV is banked between
25 // the Security states.
26 ispriv = CurrentModeIsPrivileged(secureMpu);
27 else
28 ns = !secure || sAttrib.ns;
29 secureMpu = secure;
30 else
31 ns = TRUE;
32 secureMpu = FALSE;
33
34 // Getting memory attribute information from MPU. Note that NS information
35 // in the memory attribute is set by SAU/IDAU and is updated after getting
36 // attribute values from MPU.
37 (result.memattrs, perms) = MPUCheck(address, acctype, ispriv, secureMpu);
38 // Updating NS information got from SAU/IDAU in memory attributes
39 result.memattrs.NS = ns;
40
41 // Generate UNALIGNED UsageFault exception if access to Device memory is unaligned.
42 if !aligned && result.memattrs.memtype == MemType_Device && perms.apValid == TRUE then
43 UFSR.UNALIGNED = '1';
44 excInfo = CreateException(UsageFault, FALSE, boolean UNKNOWN);
45
46 if excInfo.fault == NoFault && HaveSecurityExt() then
47 // Check if there is a SAU/IDAU violation and, if so, update the fault informations
48 raiseSecFault = FALSE;
49 if isInstrfetch then
50 if secure then
51 if sAttrib.ns then
52 // Invalid exit from the Secure state
53 SFSR.INVTRAN = '1';
54 raiseSecFault = TRUE;
55 else
56 if !sAttrib.ns && !sAttrib.nsc then
57 // Invalid entry to the Secure state
58 SFSR.INVEP = '1';
59 raiseSecFault = TRUE;
60 else
61 if !secure && !sAttrib.ns then
62 // Vector table faults don't generate SFAR/SFSR syndrome info. They are

```

```

62 // reported via HFSR.VECTBL which is not set here.
63 if HaveMainExt() && acctype != AccType_VECTABLE then
64 if acctype == AccType_LAZYFP then
65 SFSR.LSPERR = '1';
66 else
67 SFSR.AUVIOL = '1';
68 SFSR.SFARVALID = '1';
69 SFAR = address;
70 // If Main Extension is not implemented the fault always escalates to a
71 // HardFault.
72 raiseSecFault = TRUE;
73
74 if raiseSecFault then
75 excInfo = CreateException(SecureFault, TRUE, TRUE);
76
77 result.paddress = address;
78 result.accattrs.iswrite = iswrite;
79 result.accattrs.ispriv = ispriv;
80 result.accattrs.acctype = acctype;
81
82 if excInfo.fault == NoFault then
83 excInfo = CheckPermission(perms, address, acctype, iswrite, ispriv, secureMpu);
84
85 return (excInfo, result);

```

### E2.1.307 ValidateExceptionReturn

```

1 // ValidateExceptionReturn()
2 // =====
3
4 (ExcInfo, EXC_RETURN_Type) ValidateExceptionReturn(EXC_RETURN_Type excReturn, integer
5 returningExceptionNumber)
6 boolean error = FALSE;
7 assert CurrentMode() == PMode_Handler;
8 if !IsOnes(excReturn[23:7]) || excReturn[1] != '0' then
9 UNPREDICTABLE;
10 if !HaveFPEExt() && excReturn.FType == '0' then
11 UNPREDICTABLE;
12 if !HaveSecurityExt() && (excReturn.S == '1' ||
13 excReturn.ES == '1' ||
14 excReturn.DCRS == '0') then
15 UNPREDICTABLE;
16
17 // Security specific validation
18 if HaveSecurityExt() then
19 // If exception return is an invalid attempt to return from Non-secure
20 // state with EXC_RETURN.ES set as '1', then a SecureFault is raised
21 exceptionWasSecure = excReturn.ES == '1';
22 if CurrentState == SecurityState_NonSecure && excReturn.ES == '1' then
23 error = TRUE;
24 // excReturn.ES is used below to control which exception to
25 // deactivate, and which CONTROL.SPSEL to update. Force it to the
26 // correct value so the code below functions correctly even if the
27 // Non-secure state returned an invalid excReturn value.
28 // Similarly the exception to deactivate below is actually Non-secure
29 excReturn.ES = '0';
30 exceptionWasSecure = FALSE;
31
32 // Check DCRS bit not used in for Non-secure exceptions
33 if !exceptionWasSecure && excReturn.DCRS == '0' then
34 error = TRUE;
35
36 if error then
37 SFSR.INVER = '1';
38 exceptionNumber = SecureFault;
39 else
40 exceptionWasSecure = FALSE;

```



```

41 // check returning from an inactive handler
42 if !error then
43 if !IsActiveForState(returningExceptionNumber, exceptionWasSecure) then
44 error = TRUE;
45 if HaveMainExt() then
46 UFSR.INVPC = '1';
47 exceptionNumber = UsageFault;
48 else
49 exceptionNumber = HardFault;
50
51 if error then
52 DeActivate(returningExceptionNumber, exceptionWasSecure);
53 if HaveSecurityExt() && exceptionWasSecure then
54 CONTROL_S.SPSEL = excReturn.SPSEL;
55 else
56 CONTROL_NS.SPSEL = excReturn.SPSEL;
57 // Escalates to HardFault if requested fault is disabled, or has
58 // insufficient priority, or if Main Extension is not implemented
59 excInfo = CreateException(exceptionNumber, FALSE, boolean UNKNOWN);
60 else
61 excInfo = DefaultExcInfo();
62 return (excInfo, excReturn);

```

### E2.1.308 Vector

```

1 // Vector[]
2 // =====
3
4 (ExcInfo, bits(32)) Vector(integer exceptionNumber, boolean isSecure)
5 // Calculate the address of the entry in the vector table
6 vtor = if isSecure then VTOR_S else VTOR_NS;
7 addr = (vtor.TBLOFF:'0000000') + 4 * exceptionNumber;
8 // Fetch the vector with the correct privilege and security
9 (exc, vector) = MemA_with_priv_security(addr, 4, AccType_VECTABLE, TRUE, isSecure, TRUE);
10 // Faults that prevent the vector being fetched are terminal and prevent
11 // the exception being entered. They are therefore treated as HardFaults
12 if exc.fault != NoFault then
13 exc.isTerminal = TRUE;
14 exc.fault = HardFault;
15 exc.isSecure = exc.isSecure || AIRCR.BFHFNMINS == '0';
16 HFSR.VECTBL = '1';
17 return (exc, vector);

```

### E2.1.309 VFPExcBarrier

```

1 // VFPExcBarrier
2 // =====
3 // Ensures that all floating-point exception processing has completed
4
5 VFPExcBarrier();

```

### E2.1.310 VFPEexpandImm

```

1 // VFPEexpandImm()
2 // =====
3
4 bits(N) VFPEexpandImm(bits(8) imm8, integer N)
5 assert N IN {32, 64};
6 integer E = if N == 32 then 8 else 11;
7 constant integer F = N - E - 1;
8 bits(E-2) exp;
9 bits(F+2) frac;
10 sign = imm8[7];
11 exp = NOT(imm8[6]):Replicate(imm8[6], E-3);
12 frac = imm8[5:0]:Zeros(F-4);
13 return sign : exp : frac;

```

**E2.1.311 VFPNegMul**

```

1 // Different types of floating-point multiply and negate operations
2
3 enumeration VFPNegMul {VFPNegMul_VNMLA, VFPNegMul_VNMLS, VFPNegMul_VNMUL};

```

**E2.1.312 VFPSmallRegisterBank**

```

1 // VFPSmallRegisterBank
2 // =====
3 // Returns TRUE if Floating Point implementation provides access only to
4 // 16 double-precision registers
5
6 boolean VFPSmallRegisterBank();

```

**E2.1.313 WaitForEvent**

```

1 // WaitForEvent
2 // =====
3 // Optionally suspends execution until a WFE wakeup event or reset occurs,
4 // or until some earlier time if the implementation chooses
5
6 WaitForEvent();

```

**E2.1.314 WaitForInterrupt**

```

1 // WaitForInterrupt
2 // =====
3 // Optionally suspends execution until a WFI wakeup event or reset occurs, or
4 // until some earlier time if the implementation chooses
5
6 WaitForInterrupt();

```

**E2.1.315 ZeroExtend**

```

1 // ZeroExtend()
2 // =====
3
4 bits(N) ZeroExtend(bits(M) x, integer N)
5 assert N >= M;
6 return Zeros(N-M) : x;
7
8 bits(N) ZeroExtend(bits(M) x)
9 return ZeroExtend(x, N);

```

**E2.1.316 Zeros**

```

1 // Zeros()
2 // =====
3
4 bits(N) Zeros(integer N)
5 return Replicate('0', N);
6
7 bits(N) Zeros()
8 return Zeros(N);

```

**Part F**  
**Debug Packet Protocols**

## Chapter F1

# ITM and DWT Packet Protocol Specification

This chapter describes the protocol for packets that send the data generated by the ITM and DWT to an external debugger. It contains the following sections:

- [About the ITM and DWT packets.](#)
- [Alphabetical list of DWT and ITM packets.](#)

## F1.1 About the ITM and DWT packets

The following sections give an overview of the ITM and DWT packets and how the TPIU transmits them:

- [Uses of ITM and DWT packets](#)
- [ITM and DWT protocol packet headers](#)
- [Packet transmission by the trace sink](#)

### Note

This chapter describes packet transmission by a trace sink such as a TPIU. The ITM can send packets to any suitable trace sink. Regardless of the actual trace sink used, the ITM formats the packets as described in this chapter.

### F1.1.1 Uses of ITM and DWT packets

The ITM sends a packet to the trace sink when:

- Software writes to a stimulus register. This generates a [Instrumentation packet](#).
- The hardware generates a Protocol packet. Protocol packets include timestamps and synchronization packets.
- It receives a packet from the DWT, for forwarding to the trace sink.

The DWT sends a packet to the ITM for forwarding to the trace sink when:

- A DWT comparator matches and generates one or more Data Trace packets.
- It samples the PC.
- One of the performance profile counters wraps.

This chapter describes the packet protocol used.

### F1.1.2 ITM and DWT protocol packet headers

| [7] | [6]                  | [5] | [4] | [3] | [2] | [1]   | [0] | Description                                    |
|-----|----------------------|-----|-----|-----|-----|-------|-----|------------------------------------------------|
| 0   | 0                    | 0   | 0   | 0   | 0   | 0     | 0   | <a href="#">Synchronization packet</a>         |
| 0   | 1                    | 1   | 1   | 0   | 0   | 0     | 0   | <a href="#">Overflow packet</a>                |
| 0   | ≠0b000<br>&& ≠ 0b111 |     |     | 0   | 0   | 0     | 0   | <a href="#">Local Timestamp 2 packet</a>       |
| 1   | 0                    | 0   | 1   | 0   | 1   | 0     | 0   | <a href="#">Global Timestamp 1 packet</a>      |
| 1   | 0                    | 1   | 1   | 0   | 1   | 0     | 0   | <a href="#">Global Timestamp 2 packet</a>      |
| 1   | 1                    | x   | x   | 0   | 0   | 0     | 0   | <a href="#">Local Timestamp 1 packet</a>       |
| x   | x                    | x   | x   | 1   | x   | 0     | 0   | <a href="#">Extension Packet</a>               |
| 0   | 0                    | 0   | 0   | 0   | 1   | 0     | 1   | <a href="#">Event Counter Packet</a>           |
| 0   | 1                    | x   | x   | 0   | 1   | ≠0b00 |     | <a href="#">Data Trace PC Value packet</a>     |
| 0   | 1                    | x   | x   | 1   | 1   | ≠0b00 |     | <a href="#">Data Trace Data Address packet</a> |
| 1   | 0                    | x   | x   | x   | 1   | ≠0b00 |     | <a href="#">Data Trace Data Value packet</a>   |
| x   | x                    | x   | x   | x   | 0   | ≠0b00 |     | <a href="#">Instrumentation packet</a>         |
| 0   | 0                    | 0   | 1   | 0   | 1   | x     | 1   | <a href="#">Periodic PC Sample packet</a>      |

### F1.1.3 Packet transmission by the trace sink

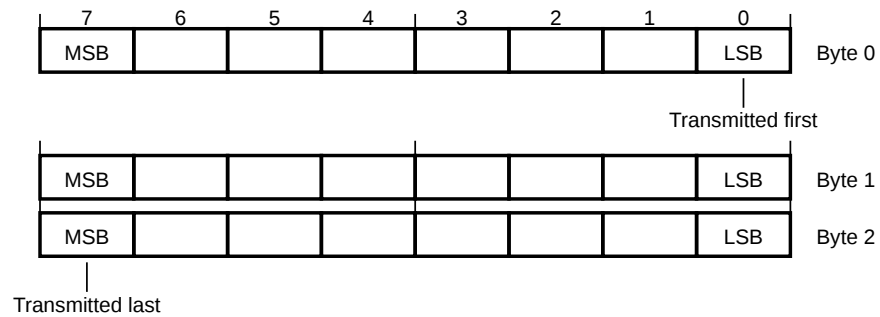
The trace sink either:

- Forms the packets into frames, as required by the *Arm@CoreSight™ Architecture Specification*.
- Transmits the packets over a serial port.

For each packet, the trace sink transmits:

- The header byte first, followed by any payload bytes.
- Each byte of the packet *least significant bit* (LSB) first.

Figures in this chapter show each packet as a sequence of bytes, with the LSB of each byte to the right and the *most significant bit* (MSB) to the left. [Convention for packet descriptions](#) shows this convention, and how it relates to data transmission for a packet with a header byte and two payload bytes.



**Figure F1.1: Convention for packet descriptions**

In some sections, the figures are split into separate figures for the header byte and payload bytes. For instance, where the number of payload bytes varies according to a field in the header.

The ITM merges the packets from the ITM and DWT with the Local and Global timestamp, Synchronization, and other Protocol packets, and forwards them to the trace sink as a single data stream. The trace sink then merges this data stream with the data from the ETM, if implemented.

## F1.2 Alphabetical list of DWT and ITM packets

### F1.2.1 Data Trace Data Address packet

The Data Trace Data Address packet characteristics are:

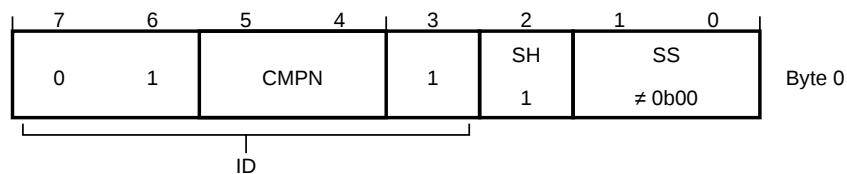
**Purpose** Indicates a DWT comparator generated a match, and the address that matched. Data Address packets are only generated for Data Address range comparator pairs. The address might be compressed. However, it is not required that Short and Medium packets are generated when the address bits match.

**Attributes** Multi-part Hardware source packet comprising:

- 8-bit header.
- 8, 16, or 32-bit payload.

#### F1.2.1.1 Data Trace Data Address packet header

The Data Trace Data Address packet header bit assignments are:



**ID, byte 0 bits [7:3]** Hardware Source packet type. Bits [7:3] discriminate between Hardware Source packet types. The defined values of this field are:

**0b01xx1** Data Trace Data Address packet.

This field reads as 0b01xx1.

**CMPN, byte 0 bits [5:4]** DWT comparator index. Defines which comparator generated a match. Data Trace Data Address packets can be compressed relative to the value in DWT\_COMP<CMPN>. The number of traced bits is indicated by the SS field. The remainder of the address bits comes from DWT\_COMP<CMPN>. Either comparator in a Data Address range comparator pair can be used.

**SH, byte 0 bit [2]** Source. The defined values of this bit are:

**1** Hardware source packet.

This bit reads as one.

**SS, byte 0 bits [1:0]** Size. The defined values of this field are:

**0b01** Short Data Address packet.

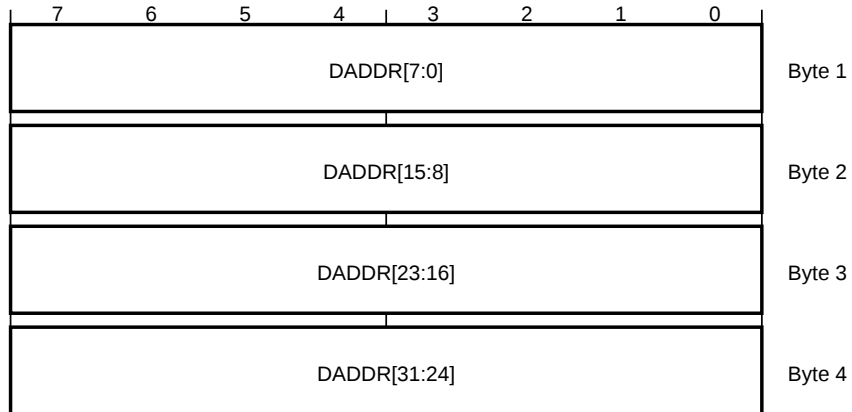
**0b10** Medium Data Address packet.

**0b11** Long Data Address packet.

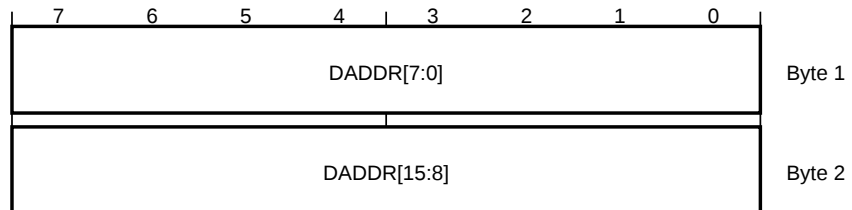
The value 0b00 encodes a Protocol packet.

### F1.2.1.2 Data Trace Data Address packet payload

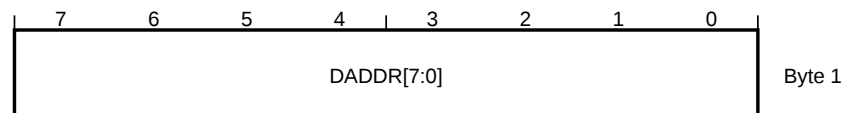
When Long Data Address packet, SS == 0b11, the Data Trace Data Address packet payload bit assignments are:



When Medium Data Address packet, SS == 0b10, the Data Trace Data Address packet payload bit assignments are:



When Short Data Address packet, SS == 0b01, the Data Trace Data Address packet payload bit assignments are:



**DADDR[31:0], bytes <4:1>, when Long Data Address packet, SS == 0b11** Data address.

**DADDR[15:0], bytes <2:1>, when Medium Data Address packet, SS == 0b10** Data address. DADDR[31:16] == DWT\_COMP<CMPN>[31:16].

**DADDR[7:0], byte <1>, when Short Data Address packet, SS == 0b01** Data address. DADDR[31:8] == DWT\_COMP<CMPN>[31:8].

### F1.2.2 Data Trace Data Value packet

The Data Trace Data Value packet characteristics are:

**Purpose** Indicates a DWT comparator generated a match, and the value that matched.

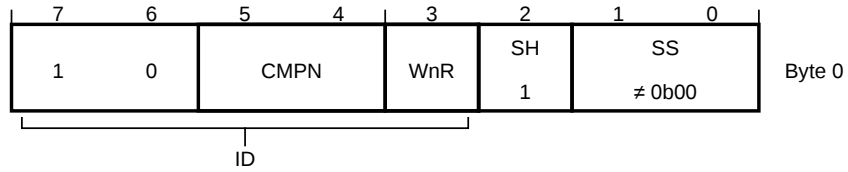
**Attributes** Multi-part Hardware source packet comprising:

- 8-bit header.
- 8, 16, or 32-bit payload.



### F1.2.2.1 Data Trace Data Value packet header

The Data Trace Data Value packet header bit assignments are:



**ID, byte 0 bits [7:3]** Hardware Source packet type. Bits [7:3] discriminate between Hardware Source packet types. The defined values of this field are:

**0b10xxx** Data Trace Data Value packet.

This field reads as 0b10xxx.

**CMPN, byte 0 bits [5:4]** DWT comparator index. Defines which comparator generated a match.

**WnR, byte 0 bit [3]** Write-not-read. The defined values of this bit are:

**0** Read.

**1** Write.

**SH, byte 0 bit [2]** Source. The defined values of this bit are:

**1** Hardware source packet.

This bit reads as one.

**SS, byte 0 bits [1:0]** Size. The defined values of this field are:

**0b01** Byte Data Value packet.

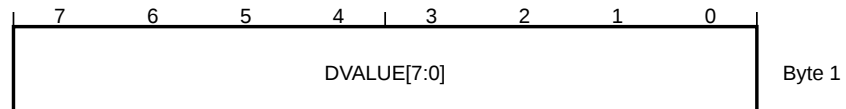
**0b10** Halfword Data Value packet.

**0b11** Word Data Value packet.

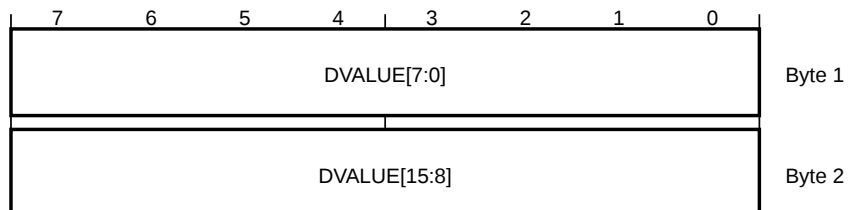
The value 0b00 encodes a Protocol packet.

### F1.2.2.2 Data Trace Data Value packet payload

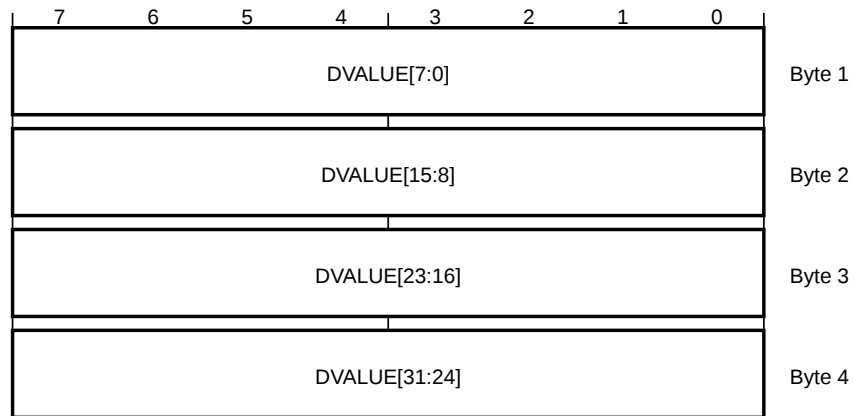
When Byte Data Value packet, SS == 0b01, the Data Trace Data Value packet payload bit assignments are:



When Halfword Data Value packet, SS == 0b10, the Data Trace Data Value packet payload bit assignments are:



When Word Data Value packet, SS == 0b11, the Data Trace Data Value packet payload bit assignments are:



**DVALUE[31:0], bytes <4:1>, when Word Data Value packet, SS == 0b11** Word data value.

**DVALUE[15:0], byte 1 bits [15:0], when Halfword Data Value packet, SS == 0b10** Halfword data value.

**DVALUE[7:0], byte <1>, when Byte Data Value packet, SS == 0b01** Byte data value.

### F1.2.3 Data Trace Match packet

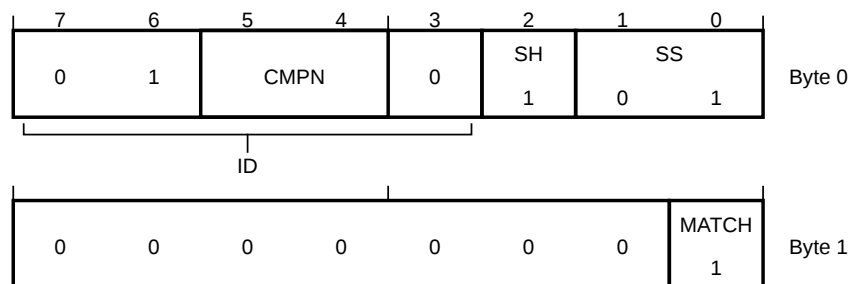
The Data Trace Match packet characteristics are:

**Purpose** Indicates a DWT comparator generated a match.

**Attributes** 16-bit Hardware source packet.

#### Field descriptions

The Data Trace Match packet bit assignments are:



**0b01xx0** [Data Trace PC Value packet](#) or Data Trace Match packet.

Bit [0] of byte 1 discriminates between the [Data Trace PC Value packet](#) and the Data Trace Match packet.

This field reads as 0b01xx0.

**CMPN, byte 0 bits [5:4]** DWT comparator index. Defines which comparator generated a match.

**SH, byte 0 bit [2]** Source. The defined values of this bit are:

1 Hardware source packet.

This bit reads as one.

**SS, byte 0 bits [1:0]** Size. The defined values of this field are:

**0b01** Source packet, 1-byte payload, 2-byte packet.

The value 0b00 encodes a Protocol packet. All other values are reserved.

This field reads as 0b01.

## F1.2.4 Data Trace PC Value packet

The Data Trace PC Value packet characteristics are:

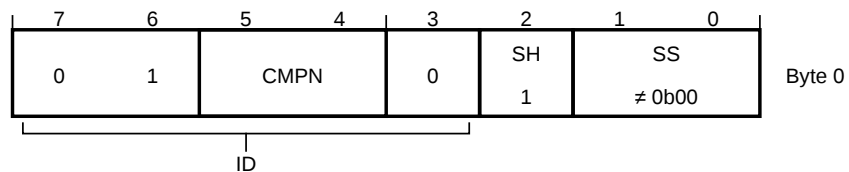
**Purpose** Indicates a DWT comparator generated a match, and the address of the instruction that matched. The address might be compressed. However, it is not required that Short and Medium packets are generated when the address bits match.

**Attributes** Multi-part Hardware source packet comprising:

- 8-bit header.
- 8, 16, or 32-bit payload.

### F1.2.4.1 Data Trace PC Value packet header

The Data Trace PC Value packet header bit assignments are:



**ID, byte 0 bits [7:3]** Hardware Source packet type. Bits [7:3] discriminate between Hardware Source packet types. The defined values of this field are:

**0b01xx0** Data Trace PC Value packet or [Data Trace Match packet](#).

Bit [0] of byte 1 discriminates between the Data Trace PC Value packet and the [Data Trace Match packet](#).

This field reads as 0b01xx0.

**CMPN, byte 0 bits [5:4]** DWT comparator index. Defines which comparator generated a match. Data Trace PC Value packets can be compressed relative to the value in DWT\_COMP<CMPN>. The number of traced bits is indicated by the SS field. The remainder of the address bits comes from DWT\_COMP<CMPN>. Either comparator in an Instruction Address range comparator pair can be used.

**SH, byte 0 bit [2]** Source. The defined values of this bit are:

1 Hardware source packet.

This bit reads as one.

**SS, byte 0 bits [1:0]** Size. The defined values of this field are:

0b01 Short PC Value packet.

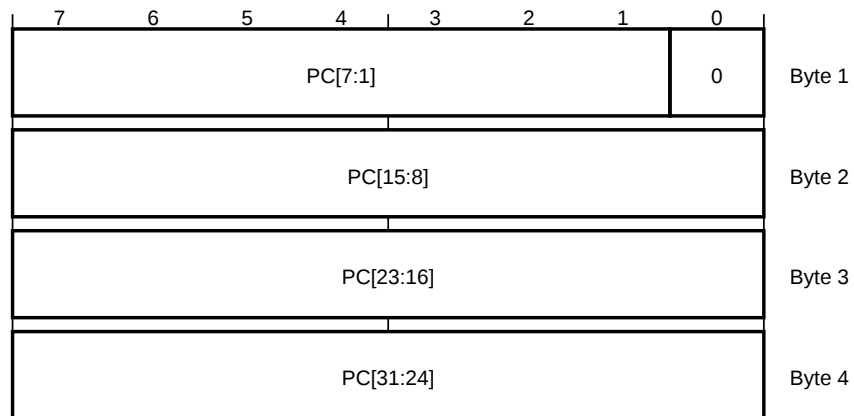
0b10 Medium PC Value packet.

0b11 Long PC Value packet.

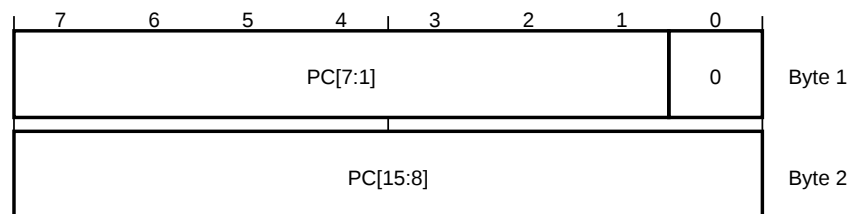
The value 0b00 encodes a Protocol packet.

### F1.2.4.2 Data Trace PC Value packet payload

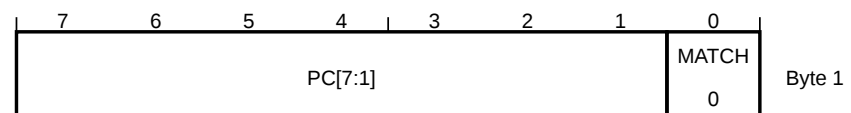
When Long PC Value packet, SS == 0b11, the Data Trace PC Value packet payload bit assignments are:



When Medium PC Value packet, SS == 0b10, the Data Trace PC Value packet payload bit assignments are:



When Short PC Value packet, SS == 0b01, the Data Trace PC Value packet payload bit assignments are:



**PC[31:1], bytes <4:2>, byte 1 bits [7:1], when Long PC Value packet, SS == 0b11** Instruction address.

**PC[15:1], byte <2>, byte 1 bits [7:1], when Medium PC Value packet, SS == 0b10** Instruction address.  
 PC[31:16] == DWT\_COMP<CMPN>[31:16].

**PC[7:1], byte 1 bits [7:1], when Short PC Value packet, SS == 0b01** Instruction address. PC[31:8] == DWT\_COMP<CMPN>[31:8].

**MATCH, byte 1 bit [0]** **Data Trace Match packet.** Discriminates between the Data Trace PC Value packet and the **Data Trace Match packet**. The defined values of this bit are:

**0** Data Trace PC Value packet.

This bit reads as zero.

### F1.2.5 Event Counter packet

The Event Counter packet characteristics are:

**Purpose** Indicates one or more DWT counters wraps through zero.

**Attributes** 16-bit Hardware source packet.

#### Field descriptions

The Event Counter packet bit assignments are:

|    |   |   |     |      |     |       |     |        |        |
|----|---|---|-----|------|-----|-------|-----|--------|--------|
| 7  | 6 | 5 | 4   | 3    | 2   | 1     | 0   |        |        |
| ID |   |   |     | SH   | SS  |       |     | Byte 0 |        |
| 0  | 0 | 0 | 0   | 0    | 1   | 0     | 1   |        |        |
| 0  |   | 0 | Cyc | Fold | LSU | Sleep | Exc | CPI    | Byte 1 |

**Byte 1 bits [7:6]** This field reads-as-zero.

**Cyc, byte 1 bit [5]** POSTCNT timer decremented to zero. See DWT\_CTRL for more information on the POSTCNT timer.

**Fold, byte 1 bit [4]** DWT\_FOLDCNT counter wrapped from 0xFF to zero.

**LSU, byte 1 bit [3]** DWT\_LSUNCT counter wrapped from 0xFF to zero.

**Sleep, byte 1 bit [2]** DWT\_SLEPCNT counter wrapped from 0xFF to zero.

**Exc, byte 1 bit [1]** DWT\_EXCCNT counter wrapped from 0xFF to zero.

**CPI, byte 1 bit [0]** DWT\_CPICNT counter wrapped from 0xFF to zero.

**ID, byte 0 bits [7:3]** Hardware Source packet type. Bits [7:3] discriminate between Hardware Source packet types. The defined values of this field are:

**0b00000** Event Counter packet.

This field reads as 0b00000.

**SH, byte 0 bit [2]** Source. The defined values of this bit are:

**1** Hardware source packet.

This bit reads as one.

**SS, byte 0 bits [1:0]** Size. The defined values of this field are:

**0b01** Source packet, 1-byte payload, 2-byte packet.

The value 0b00 encodes a Protocol packet. All other values are reserved.

This field reads as 0b01.

## F1.2.6 Exception Trace packet

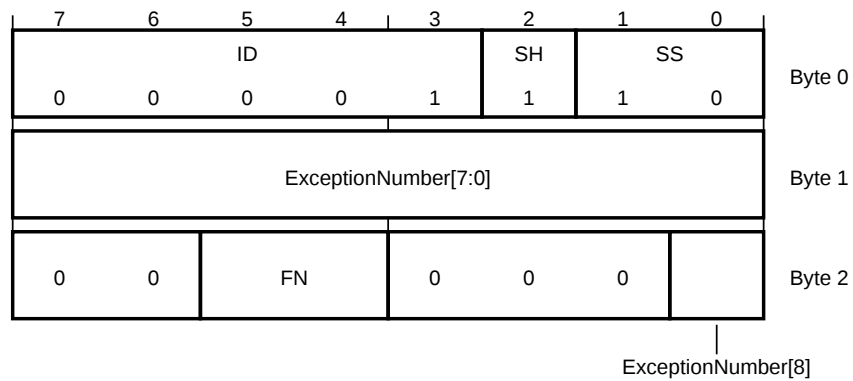
The Exception Trace packet characteristics are:

**Purpose** Indicates the PE has entered, exited or returned to an exception.

**Attributes** 24-bit Hardware source packet.

### Field descriptions

The Exception Trace packet bit assignments are:



**Byte 2 bits [7:6,3:1]** This field reads-as-zero.

**FN, byte 2 bits [5:4]** Function. The defined values of this field are:

- 0b01** Entered exception indicated by ExceptionNumber.
- 0b10** Exited exception indicated by ExceptionNumber.
- 0b11** Returned to exception indicated by ExceptionNumber.

All other values are reserved.

**ExceptionNumber, byte 2 bit [0], byte <1>** The exception number.

**ID, byte 0 bits [7:3]** Hardware Source packet type. Bits [7:3] discriminate between Hardware Source packet types. The defined values of this field are:

- 0b00001** Exception Trace packet.

This field reads as 0b00001.

**SH, byte 0 bit [2]** Source. The defined values of this bit are:

- 1** Hardware source packet.

This bit reads as one.

**SS, byte 0 bits [1:0]** Size. The defined values of this field are:

- 0b10** Source packet, 2-byte payload, 3-byte packet.

The value 0b00 encodes a Protocol packet. All other values are reserved.

This field reads as 0b10.

### F1.2.7 Extension packet

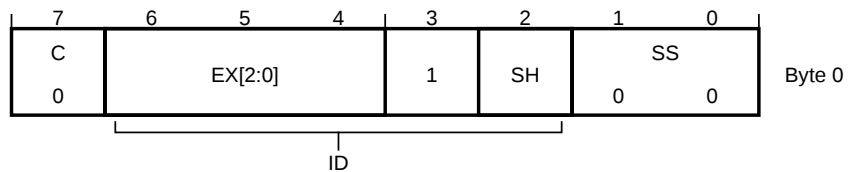
The Extension packet characteristics are:

**Purpose** An Extension packet provides additional information about the identified source. The amount of information required determines the number of payload bytes, 0-4. The architecture only defines one use of the Extension packet, to provide a Stimulus port page number. For this use, SH == 0, and a single byte Extension packet is emitted.

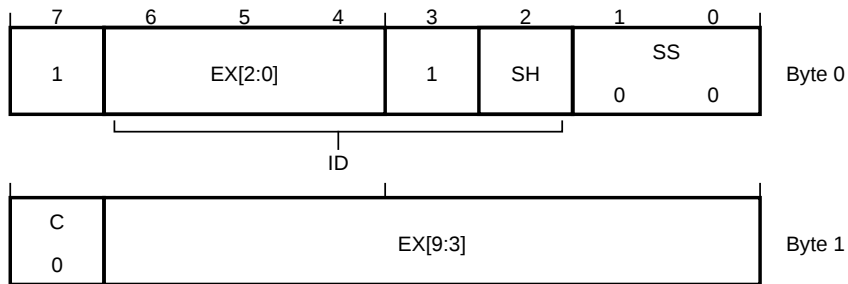
**Attributes** 8, 16, 24, 32, or 40-bit Protocol packet.

#### Field descriptions

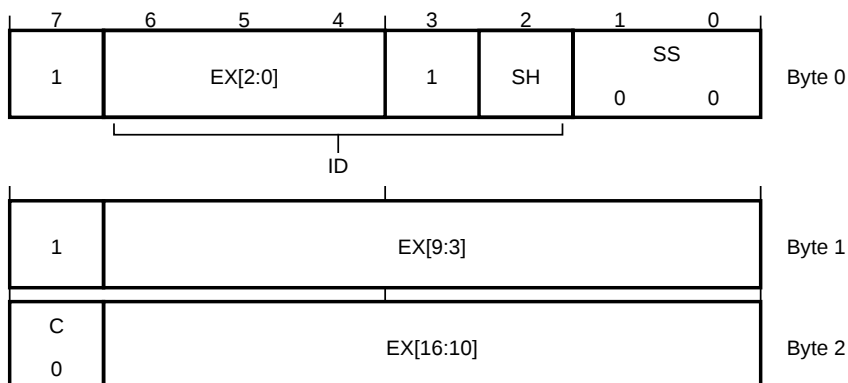
When 1-byte packet, the Extension packet bit assignments are:



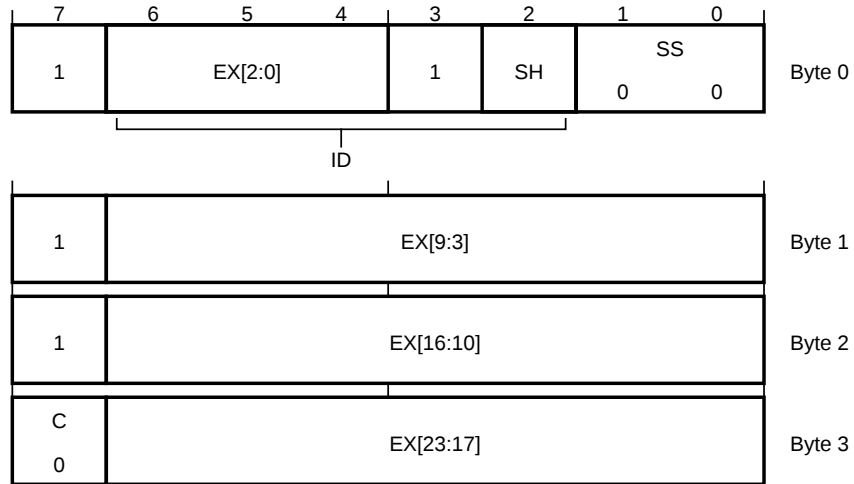
When 2-byte packet, the Extension packet bit assignments are:



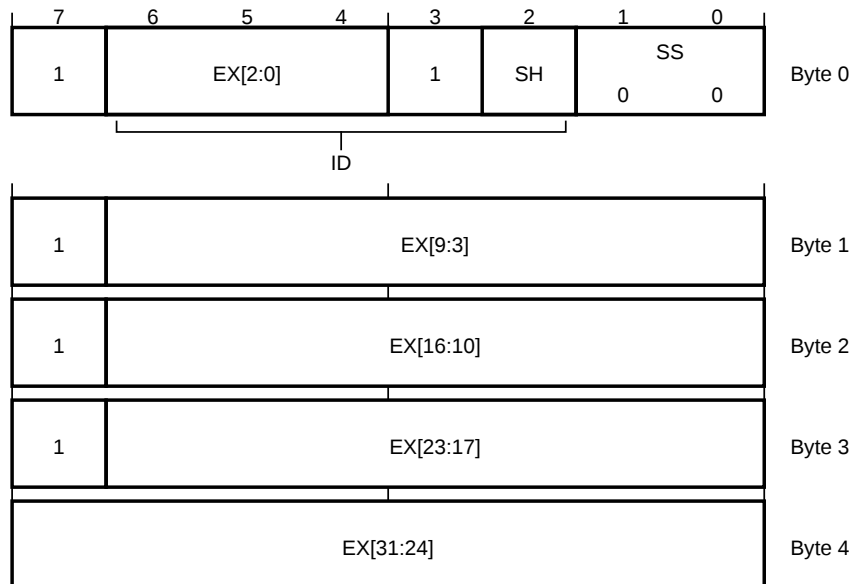
When 3-byte packet, the Extension packet bit assignments are:



When 4-byte packet, the Extension packet bit assignments are:



When 5-byte packet, the Extension packet bit assignments are:



**EX, byte <4>, byte 3 bits [6:0], byte 2 bits [6:0], byte 1 bits [6:0], byte 0 bits [6:4]** Extension information. If SH == 1, then EX defines PAGE, the Stimulus port page number.

This is a 32-bit field. If the Extension packet is shorter than 5 bytes, the most significant bits are zero.

**C, byte 3 bit [7], byte 2 bit [7], byte 1 bit [7], byte 0 bit [7]** Continuation bit. The defined values of this field are:

- 0 Last byte of the packet.
- 1 Another byte follows.

**ID, byte 0 bits [6:2]** Protocol packet type. Bits [6:2] discriminate between Protocol packet types. The defined values of this field are:

**0bxxx1x** Extension packet.

This field reads as 0bxxx1x.

**SH, byte 0 bit [2]** Source. The defined values of this bit are:



- 0 Extension packet for [Instrumentation packet](#).
- 1 Extension packet for Hardware source packet.

**SS, byte 0 bits [1:0]** Packet type. The defined values of this field are:

**0b00** Protocol packet.

Other values encode different sizes of Hardware and Software source packets. This field reads as 0b00.

### F1.2.8 Global Timestamp 1 packet

The Global Timestamp 1 packet characteristics are:

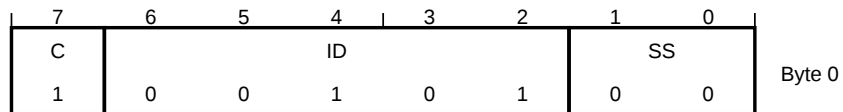
**Purpose** Contains the least significant bits of the global timestamp value. The ITM might compress this value if it is not generating a full timestamp by omitting significant bits if they are unchanged from the previous timestamp value.

**Attributes** Multi-part Protocol packet comprising:

- 8-bit header.
- 8, 16, 24, or 32-bit payload.

#### F1.2.8.1 Global Timestamp 1 packet header

The Global Timestamp 1 packet header bit assignments are:



**C, byte 0 bit [7]** Continuation bit. This bit reads as one.

**ID, byte 0 bits [6:2]** Protocol packet type. Bits [6:2] discriminate between Protocol packet types. The defined values of this field are:

**0b00101** Global Timestamp 1 packet.

This field reads as 0b00101.

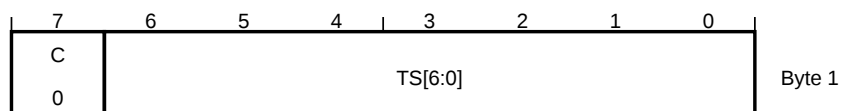
**SS, byte 0 bits [1:0]** Packet type. The defined values of this field are:

**0b00** Protocol packet.

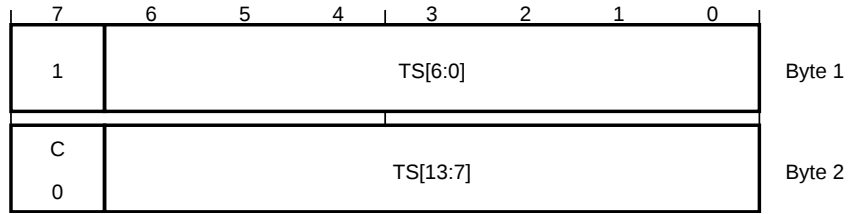
Other values encode different sizes of Hardware and Software source packets. This field reads as 0b00.

#### F1.2.8.2 Global Timestamp 1 packet payload

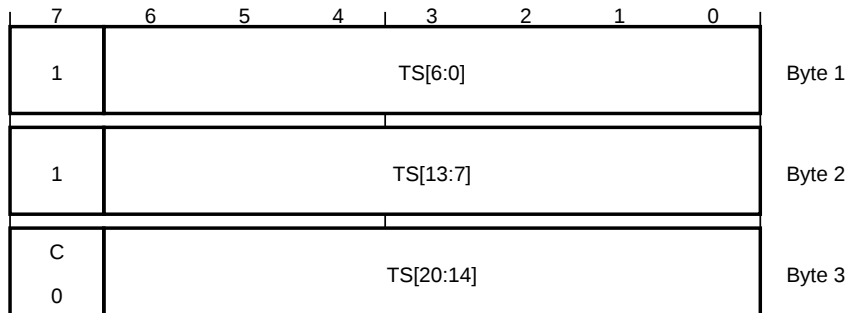
When 7-bit timestamp, the Global Timestamp 1 packet payload bit assignments are:



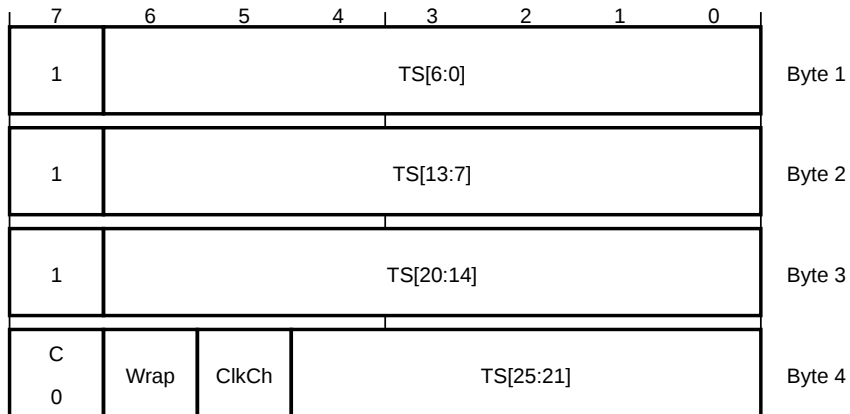
When 14-bit timestamp, the Global Timestamp 1 packet payload bit assignments are:



When 21-bit timestamp, the Global Timestamp 1 packet payload bit assignments are:



When 26-bit or full timestamp, the Global Timestamp 1 packet payload bit assignments are:



**C, byte 4 bit [7], byte 3 bit [7], byte 2 bit [7], byte 1 bit [7]** Continuation bit. The defined values of this field are:

- 0 Last byte of the packet.
- 1 Another byte follows.

**Wrap, byte 4 bit [6], when 26-bit or full timestamp** Wrapped. The defined values of this bit are:

- 0 The value of global timestamp bits TS[47:26] or TS[63:26] have not changed since the last [Global Timestamp 2 packet](#) output by the ITM.
- 1 The value of global timestamp bits TS[47:26] or TS[63:26] have changed since the last [Global Timestamp 2 packet](#) output by the ITM.

**ClkCh, byte 4 bit [5], when 26-bit or full timestamp** Clock change. The defined values of this bit are:

- 0 The system has not asserted the clock change input to the processor since the last time the ITM generated a Global Timestamp packet.

- 1 The system has asserted the clock change input to the processor since the last time the ITM generated a Global Timestamp packet.

**Note**

When the clock change input to the processor is asserted, the ITM must output a full 48-bit or 64-bit global timestamp value.

**TS[25:0], byte 4 bits [4:0], byte 3 bits [6:0], byte 2 bits [6:0], byte 1 bits [6:0]** Global Timestamp. The timestamp is 64 or 48 bits. If the Global Timestamp 1 packet is shorter than 5 bytes, the most-significant bits of the timestamp have not changed since the last Global Timestamp 1 packet output by the ITM. If the Global Timestamp 1 packet is 5 bytes, the Wrap bit defines whether most-significant bits have unchanged since the last [Global Timestamp 2 packet](#) output by the ITM.

### F1.2.9 Global Timestamp 2 packet

The Global Timestamp 2 packet characteristics are:

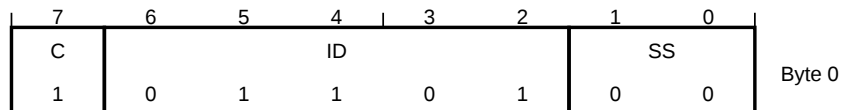
**Purpose** Provides the most significant bits of a full 48 or 64-bit timestamp.

**Attributes** Multi-part Protocol packet comprising:

- 8-bit header.
- 32 or 48-bit payload.

#### F1.2.9.1 Global Timestamp 2 packet header

The Global Timestamp 2 packet header bit assignments are:



**C, byte 0 bit [7]** Continuation bit. This bit reads as one.

**ID, byte 0 bits [6:2]** Protocol packet type. Bits [6:2] discriminate between Protocol packet types. The defined values of this field are:

**0b01101** Global Timestamp 2 packet.

This field reads as 0b01101.

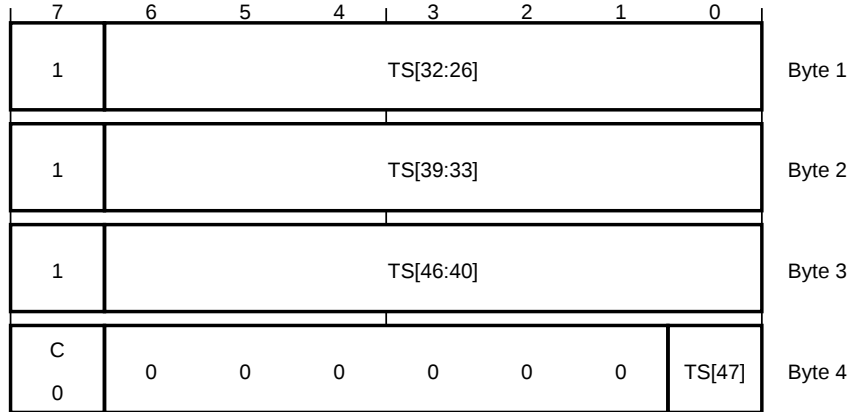
**SS, byte 0 bits [1:0]** Packet type. The defined values of this field are:

**0b00** Protocol packet.

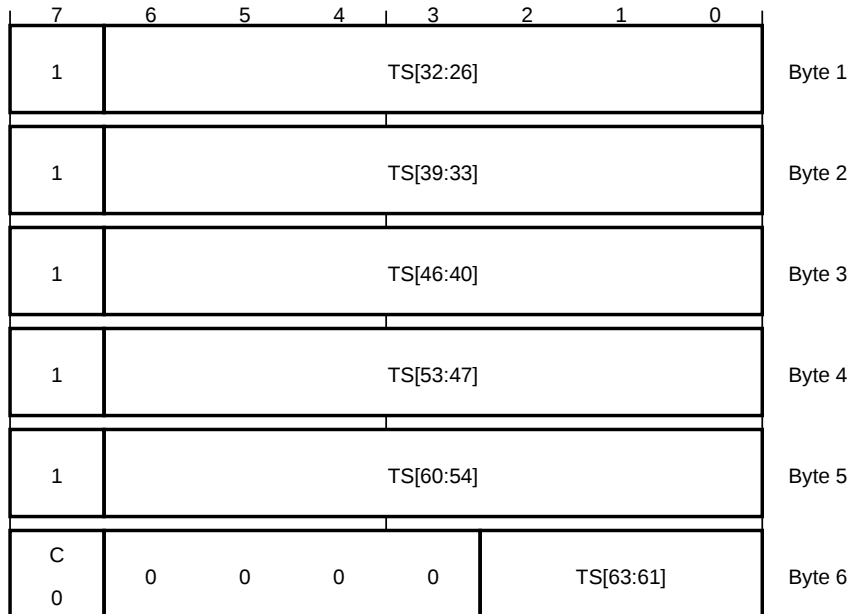
Other values encode different sizes of Hardware and Software source packets. This field reads as 0b00.

#### F1.2.9.2 Global Timestamp 2 packet payload

When 48-bit Global Timestamp 2 packet, the Global Timestamp 2 packet payload bit assignments are:



When 64-bit Global Timestamp 2 packet, the Global Timestamp 2 packet payload bit assignments are:



**C, byte 6 bit [7], byte 5 bit [7], byte 4 bit [7], byte 3 bit [7], byte 2 bit [7], byte 1 bit [7] Continuation bit.**

The defined values of this field are:

0 Last byte of the packet.

1 Another byte follows.

**Byte 6 bits [6:3], when 64-bit Global Timestamp 2 packet** This field reads-as-zero.

**Byte 4 bits [6:1], when 48-bit Global Timestamp 2 packet** This field reads-as-zero.

**TS[47:26] , byte 4 bit [0], byte 3 bits [6:0], byte 2 bits [6:0], byte 1 bits [6:0], when 48-bit Global Timestamp 2 packet**

Most significant bits of the Global Timestamp.

**TS[63:26] , byte 6 bits [2:0], byte 5 bits [6:0], byte 4 bits [6:0], byte 3 bits [6:0], byte 2 bits [6:0], byte 1 bits [6:0], when 64-bit Global Timestamp 2 packet**

Most significant bits of the Global Timestamp.

### F1.2.10 Instrumentation packet

The Instrumentation packet characteristics are:

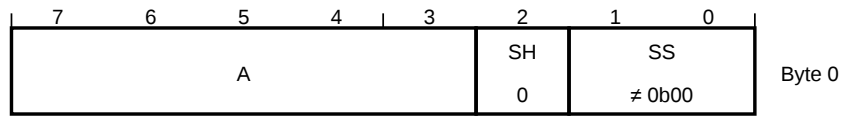
**Purpose** A software write to an ITM stimulus port generates an Instrumentation packet.

**Attributes** Multi-part Software source packet comprising:

- 8-bit header.
- 8, 16, or 32-bit payload.

#### F1.2.10.1 Instrumentation packet header

The Instrumentation packet header bit assignments are:



**A, byte 0 bits [7:3]** Port number, 0-31. If an ITM implementation supports more than 32 stimulus ports, paging is used to indicate the stimulus port number. The stimulus port number is defined by the A field of the Instrumentation packet, byte 0 bits [7:3]:

$$\text{Stimulus port number} = (\text{Page number} \times 32) + A[4:0]$$

**SH, byte 0 bit [2]** Source. The defined values of this bit are:

0 Instrumentation packet (Software source).

This bit reads as zero.

**SS, byte 0 bits [1:0]** Size. The defined values of this field are:

0b01 Byte Instrumentation packet.

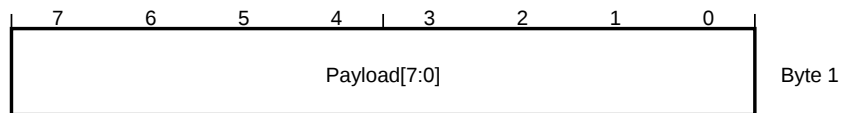
0b10 Halfword Instrumentation packet.

0b11 Word Instrumentation packet.

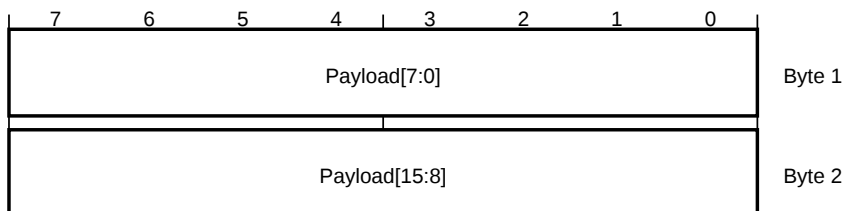
The value 0b00 encodes a Protocol packet.

#### F1.2.10.2 Instrumentation packet payload

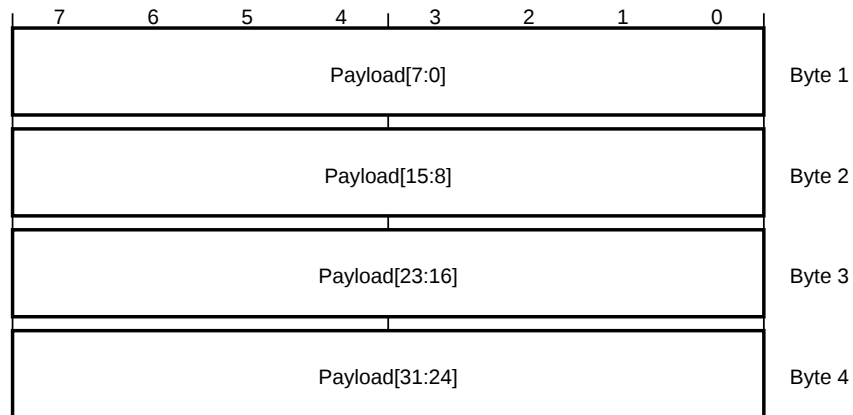
When Byte Instrumentation packet, SS == 0b01, the Instrumentation packet payload bit assignments are:



When Halfword Instrumentation packet, SS == 0b10, the Instrumentation packet payload bit assignments are:



When Word Instrumentation packet, SS == 0b11, the Instrumentation packet payload bit assignments are:



**Payload[31:0], bytes <4:1>, when Word Instrumentation packet, SS == 0b11** Payload value.

**Payload[15:0], byte 1 bits [15:0], when Halfword Instrumentation packet, SS == 0b10** Payload value.

**Payload[7:0], byte <1>, when Byte Instrumentation packet, SS == 0b01** Payload value.

### F1.2.11 Local Timestamp 1 packet

The Local Timestamp 1 packet characteristics are:

**Purpose** A Local Timestamp 1 packet encodes timestamp information, for generic control and synchronization, based on a timestamp counter in the ITM. To reduce the trace bandwidth:

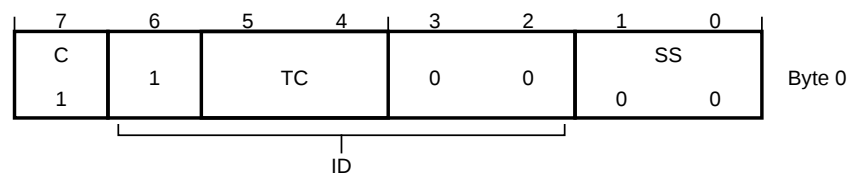
- The local timestamping scheme uses delta timestamps. Whenever the ITM outputs a Local timestamp packet, it clears its timestamp counter to zero, meaning each local timestamp value gives the interval since the generation of the previous Local timestamp packet.
- The Local Timestamp 1 packet length, 1-5 bytes, depends on the timestamp value.
- If the ITM outputs the local timestamp synchronously to the corresponding ITM or DWT data, and the timestamp value is in the range 1-6, the ITM uses the [Local Timestamp 2 packet](#).

**Attributes** Multi-part Protocol packet comprising:

- 8-bit header.
- 8, 16, 24, or 32-bit payload.

#### F1.2.11.1 Local Timestamp 1 packet header

The Local Timestamp 1 packet header bit assignments are:



**C, byte 0 bit [7]** Continuation bit. This bit reads as one.

**ID, byte 0 bits [6:2]** Protocol packet type. Bits [6:2] discriminate between Protocol packet types. The defined values of this field are:

**0b1xx00** Local Timestamp 1 packet.

This field reads as 0b1xx00.

**TC, byte 0 bits [5:4]** Indicates the relationship between the generation of the Local timestamp packet and the corresponding ITM or DWT data packet. The defined values of this field are:

**0b00** The local timestamp value is synchronous to the corresponding ITM or DWT data. The value in the TS field is the timestamp counter value when the ITM or DWT packet is generated.

**0b01** The local timestamp value is delayed relative to the ITM or DWT data. The value in the TS field is the timestamp counter value when the Local timestamp packet is generated.

**Note**

The local timestamp value corresponding to the previous ITM or DWT packet is *unknown*, but must be between the previous and current local timestamp values.

**0b10** Output of the ITM or DWT packet corresponding to this Local timestamp packet is delayed relative to the associated event. The value in the TS field is the timestamp counter value when the ITM or DWT packets is generated.

This encoding indicates that the ITM or DWT packet was delayed relative to other trace output packets.

**0b11** Output of the ITM or DWT packet corresponding to this Local timestamp packet is delayed relative to the associated event, and this Local timestamp packet is delayed relative to the ITM or DWT data. This is a combination of the conditions indicated by values 0b01 and 0b10.

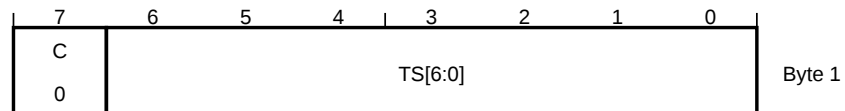
**SS, byte 0 bits [1:0]** Packet type. The defined values of this field are:

**0b00** Protocol packet.

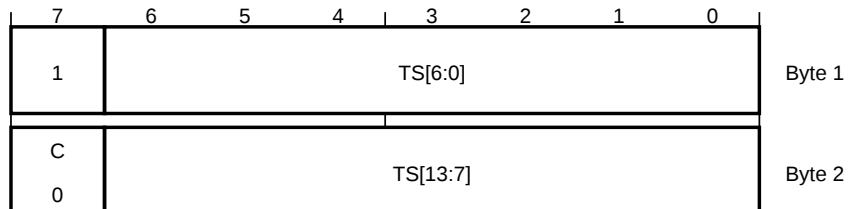
Other values encode different sizes of Hardware and Software source packets. This field reads as 0b00.

**F1.2.11.2 Local Timestamp 1 packet payload**

When 7-bit timestamp, the Local Timestamp 1 packet payload bit assignments are:



When 14-bit timestamp, the Local Timestamp 1 packet payload bit assignments are:



When 21-bit timestamp, the Local Timestamp 1 packet payload bit assignments are:



When 28-bit timestamp, the Local Timestamp 1 packet payload bit assignments are:



**C, byte 4 bit [7], byte 3 bit [7], byte 2 bit [7], byte 1 bit [7]** Continuation bit. The defined values of this field are:

- 0 Last byte of the packet.
- 1 Another byte follows.

**Byte 4 bits [6:5], when 28-bit timestamp** This field reads-as-zero.

**TS, byte 4 bits [4:0], byte 3 bits [6:0], byte 2 bits [6:0], byte 1 bits [6:0]** Local Timestamp.

The timestamp is 28 bits. If the Local Timestamp 1 packet is shorter than 5 bytes, the most significant bits of the timestamp are zero.

### F1.2.12 Local Timestamp 2 packet

The Local Timestamp 2 packet characteristics are:

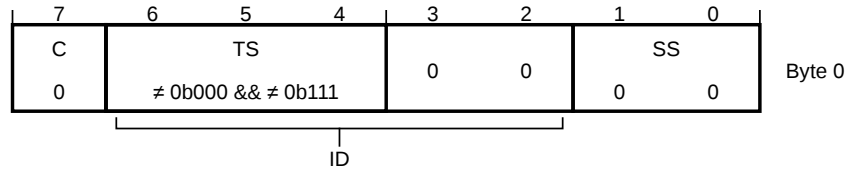
**Purpose** If the ITM outputs the Local Timestamp synchronously to the corresponding ITM or DWT data, and the required timestamp value is in the range 1-6, it uses the Local Timestamp 2 packet. For more information, see [Local Timestamp 1 packet](#).

**Attributes** 8-bit Protocol packet.

#### Field descriptions

The Local Timestamp 2 packet bit assignments are:





**C, byte 0 bit [7]** Continuation bit. This bit reads as zero.

**ID, byte 0 bits [6:2]** Protocol packet type. Bits [6:2] discriminate between Protocol packet types. The defined values of this field are:

**0b00000** See [Synchronization packet](#).

**0bxxxx00** For all other values of 0bxxxx. Local Timestamp 2 packet.

**0b11100** See [Overflow packet](#).

This field reads as 0bxxxx00.

**TS, byte 0 bits [6:4]** Local timestamp value, in the range 0b001 to 0b110.

**SS, byte 0 bits [1:0]** Packet type. The defined values of this field are:

**0b00** Protocol packet.

Other values encode different sizes of Hardware and Software source packets. This field reads as 0b00.

### F1.2.13 Overflow packet

The Overflow packet characteristics are:

**Purpose** The ITM outputs an Overflow packet if:

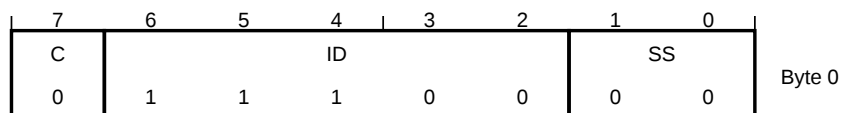
- Software writes to a Stimulus Port register when the stimulus port output buffer is full.
- The DWT attempts to generate a Hardware source packet when the DWT output buffer is full.
- The Local timestamp counter overflows.

The Overflow packet comprises a header with no payload.

**Attributes** 8-bit Protocol packet.

#### Field descriptions

The Overflow packet bit assignments are:



**C, byte 0 bit [7]** Continuation bit. This bit reads as zero.

**ID, byte 0 bits [6:2]** Protocol packet type. Bits [6:2] discriminate between Protocol packet types. The defined values of this field are:

**0b11100** Overflow packet.

This field reads as 0b11100.

**SS, byte 0 bits [1:0]** Packet type. The defined values of this field are:

**0b00** Protocol packet.

Other values encode different sizes of Hardware and Software source packets. This field reads as 0b00.

### F1.2.14 Periodic PC Sample packet

The Periodic PC Sample packet characteristics are:

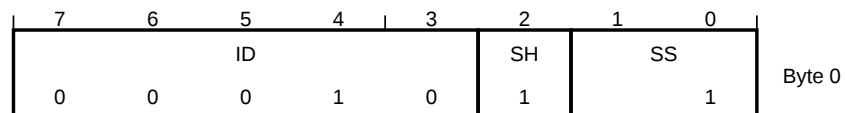
**Purpose** The DWT unit generates PC samples at fixed time intervals, with an accuracy of one clock cycle. The POSTCNT counter period determines the PC sampling interval. Software configures the [DWT\\_CTRL.CYCTAP](#) and [DWT\\_CTRL.POSTINIT](#) fields to determine how POSTCNT relates to [DWT\\_CYCCNT](#). The [DWT\\_CTRL.PCSAMPLENA](#) bit enables PC sampling.

**Attributes** Multi-part Hardware source packet comprising:

- 8-bit header.
- 8 or 32-bit payload.

#### F1.2.14.1 Periodic PC Sample packet header

The Periodic PC Sample packet header bit assignments are:



**ID, byte 0 bits [7:3]** Discriminator ID. The defined values of this field are:

**0b00010** Periodic PC Sample packet.

This field reads as 0b00010.

**SH, byte 0 bit [2]** Source. The defined values of this bit are:

**1** Hardware source packet.

This bit reads as one.

**SS, byte 0 bits [1:0]** Size. The defined values of this field are:

**0b01** Source packet, 1-byte payload, 2-byte packet.

**0b11** Source packet, 4-byte payload, 5-byte packet.

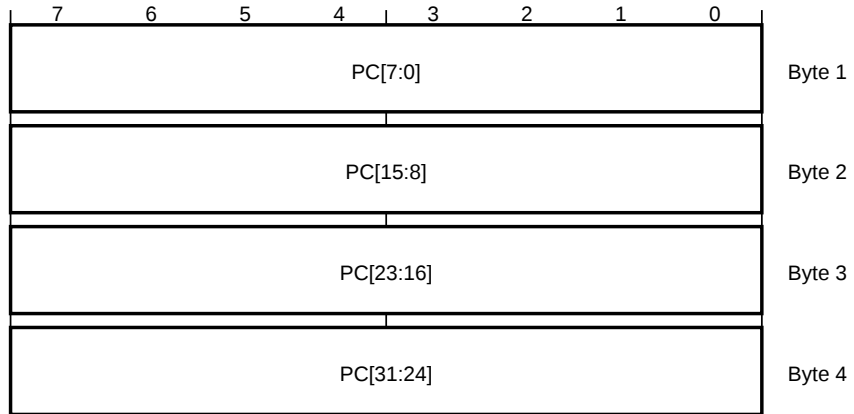
SS == 0b10 is invalid for a Periodic PC Sample packet.

The value 0b00 encodes a Protocol packet.

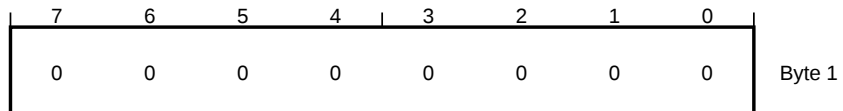
This field reads as 0bx1.

#### F1.2.14.2 Periodic PC Sample packet payload

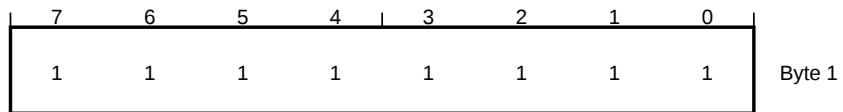
When Allowed and not sleeping, SS == 0b11, the Periodic PC Sample packet payload bit assignments are:



When Allowed and sleeping, SS == 0b01, the Periodic PC Sample packet payload bit assignments are:



When Prohibited, SS == 0b01, the Periodic PC Sample packet payload bit assignments are:



**PC, bytes <4:1>, when Allowed and not sleeping, SS == 0b11** Periodic PC sample value.

**Byte <1>, when Allowed and sleeping, SS == 0b01** This field reads as 0b00000000.

**Byte <1>, when Prohibited, SS == 0b01** This field reads as 0b11111111.

### F1.2.15 Synchronization packet

The Synchronization packet characteristics are:

**Purpose** A Synchronization packet provides a unique pattern in the bit stream. Trace capture hardware can identify this pattern and use it to identify the alignment of packet bytes in the bitstream.

**Attributes** 48-bit Protocol packet.

A Synchronization packet is at least forty-seven 0 bits followed by single 1 bit. This section describes the smallest possible Synchronization packet.

#### Field descriptions

The Synchronization packet bit assignments are:

| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |        |
|---|---|---|---|---|---|---|---|--------|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | Byte 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | Byte 1 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | Byte 2 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | Byte 3 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | Byte 4 |
| 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | Byte 5 |

**Byte 5 bit [7]** Indicates the end of the Synchronization packet. This bit reads as one.

**Byte 5 bits [6:0], bytes <4:1>** This field reads-as-zero.

**Byte <0>** This field reads as 0b00000000.

# Glossary

## AAPCS

Procedure Call Standard for the Arm Architecture.

## Address dependency

An address dependency exists when the value that is returned by a read computes the address of a subsequent access. An address dependency exists even if the value that is returned by the first read does not change the address of the second read or write.

## Addressing mode

Means a method for generating the memory address that is used by a load/store instruction.

## Aligned

A data item that is stored at an address that is exactly divisible by the highest power of 2 that divides exactly into its size in bytes. Aligned halfwords, words, and doublewords therefore have addresses that are divisible by 2, 4 and 8 respectively.

An aligned access is one where the address of the access is aligned to the size of each element of the access.

## Application Program Status Register (APSR)

The register containing those bits that deliver status information about the results of instructions, the N, Z, C, and V bits of the XPSR. In an implementation that includes the DSP extension, the APSR includes the GE bits that provide status information from DSP operations.

See also [B3.5 XPSR, APSR, IPSR, and EPSR on page 59](#).

## APSR

See Application Program Status Register.

## Architecturally executed

An instruction is architecturally executed only if it would be executed in a simple sequential execution of the program. When such an instruction has been executed and retired it has been *architecturally executed*. Any instruction that, in a simple sequential execution of a program, is treated as a NOP because it fails its condition code check, is an architecturally executed instruction.

In a PE that performs Speculative execution, an instruction is not architecturally executed if the PE discards the results of a Speculative execution.

See also [Condition code check](#), [Simple sequential execution](#).

## Architecturally UNKNOWN

An Architecturally UNKNOWN value is a value that is not defined by the architecture but must meet the requirements of the definition of UNKNOWN. Implementations can define the value of the field, but are not required to do so.

See also [IMPLEMENTATION DEFINED](#).

## Architecture tick

An atomic unit of execution. In the Armv8.0-M architecture, most instructions are considered atomic units for execution (they are either performed or not performed). The most notable exceptions are instructions that support ICI behavior.

## Associativity

See [Cache associativity](#)

### Atomicity

Describes either single-copy atomicity or multi-copy atomicity. [B5.5 Atomicity on page 149](#) defines these forms of atomicity for the Arm architecture.

See also [Multi-copy atomicity](#), [Single-copy atomicity](#).

### Attribution Unit (AU)

The combination of the Secure Attribution Unit (SAU) and the Implementation Defined Attribution Unit (IDAU).

See also [Chapter B8 The Armv8-M Protected Memory System Architecture on page 214](#).

### AU

See [Attribution unit](#).

### Background state

The state of the PE before the last (previous) preemption occurred.

### Banked register

A register that has multiple instances, with the instance that is in use depending on the PE mode, Security state, or other PE state.

### Base register

A register that is specified by a load/store instruction that is used as the base value for the address calculation for the instruction. Depending on the instruction and its addressing mode, an offset can be added to or subtracted from the base register value to form the address that is sent to memory.

### Base register Write-Back

Describes writing back a modified value to the base register used in an address calculation.

### Behaves as if

Where this manual indicates that a PE *behaves as if* a certain condition applies, all descriptions of the operation of the PE must be re-evaluated taking account of that condition, together with any other conditions that affect operation.

### Big-endian memory

Means that, for example:

- A byte or halfword at a word-aligned address is the most significant byte or halfword in the word at that address.
- A byte at a halfword-aligned address is the most significant byte in the halfword at that address.

See also [B5.3 Endianness on page 146](#), [Little-endian memory](#).

### Blocking

Describes an operation that does not permit following instructions to be executed before the operation completes.

A non-blocking operation can permit following instructions to be executed before the operation completes, and in the event of encountering an exception does not signal an exception to the PE. This enables implementations to retire following instructions while the non-blocking operation is executing, without the need to retain precise PE state.

### Branch prediction

Is where a PE selects a future execution path to fetch along. For example, after a branch instruction, the PE can choose to speculatively fetch either the instruction following the branch or the instruction at the branch target.

See also [Prefetching](#).

### Breakpoint

A debug event that is triggered by the execution of a particular instruction, which is specified by one or both of the address of the instruction and the state of the PE when the instruction is executed.

### Byte

An 8-bit data item.

### Cache associativity

The number of locations in a cache set to which an address can be assigned. Each location is identified by its *way* value.

### Cache level

The position of a cache in the cache hierarchy. In the Arm architecture, the lower numbered levels are those closest to the PE. For more information, see [B5.24 Caches on page 183](#).

### Cache line

The basic unit of storage in a cache. Its size in words is always a power of two, usually four or eight words. A cache line must be aligned to a suitable memory boundary. A *memory cache line* is a block of memory locations with the same size and alignment as a cache line. Memory cache lines are sometimes loosely called cache lines.

### Cache sets

Areas of a cache, which is divided up to simplify and speed up the process of determining whether a cache hit occurs. The number of cache sets is always a power of two. The term cache sets is a common convention for describing cache memories, and this description must not be treated as defining a property of the cache.

### Cache way

A cache way consists of one cache line from each cache set. The cache ways are indexed from 0 to (Associativity-1). Each cache line in a cache way is chosen to have the same index as the cache way. For example, cache way *n* consists of the cache line with index *n* from each cache set. The term cache way is a common convention for describing cache memories, and this description must not be treated as defining a property of the cache.

### Cache write-back granule

The maximum size of the memory that can be overwritten. In some implementations, the [CTR](#) identifies the Cache Write-Back Granule.

### Callee-saved registers

Are registers that a called procedure must preserve. To preserve a callee-saved register, the called procedure would normally either not use the register at all, or store the register to the stack during procedure entry and reload it from the stack during procedure exit.

### Caller-saved registers

Are registers that a called procedure is not required to preserve. If the calling procedure requires their values to be preserved, it must store and reload them itself.

### Coherence order

See [Coherent](#)

### Coherent

Data accesses from a set of observers to a byte in memory are coherent if accesses to that byte in memory by the members of that set of observers are consistent with there being a single total order of all writes to that byte in memory by all members of the set of observers. This single total order of all to writes to that memory location is the *coherence order* for that byte in memory.

### Condition code check

The process of determining whether a conditional instruction executes normally or is treated as a NOP. For an instruction that includes a condition code field, that field is compared with the condition flags to determine whether the instruction is executed normally. For a T32 instruction in an IT block, the value of [EPSR.IT](#) determines whether the instruction is executed normally.

See also [Condition code field](#), [Condition flags](#), [Conditional execution](#).

### Condition code field

A 4-bit field in an instruction that specifies the condition under which the instruction executes.

See also [Condition code check](#).

### Condition flags

The N, Z, C, and V bits of APSR, or XPSR. See [B3.5 XPSR, APSR, IPSR, and EPSR on page 59](#) for more information.

See also [Condition code check](#).

### Conditional execution

When a conditional instruction starts executing, if the condition code check returns TRUE, the instruction executes normally. Otherwise, it is treated as a NOP. See [C1.3 Conditional execution on page 313](#).

See also [Condition code check](#).

### Configuration

Settings that are made on reset, or immediately after reset, and normally expected to remain static throughout program execution.

### CONSTRAINED UNPREDICTABLE

Where an instruction can result in UNPREDICTABLE behavior, the Armv8 architecture specifies a narrow range of permitted behaviors. This range is the range of CONSTRAINED UNPREDICTABLE behavior. All implementations that are compliant with the architecture must follow the CONSTRAINED UNPREDICTABLE behavior within the limits defined for each particular case, and this behavior might vary.

In body text, the term CONSTRAINED UNPREDICTABLE is shown in SMALLCAPS.

See also [UNPREDICTABLE](#).

### Containable

An error that is not uncontained. A Containable error is also referred to as a Contained error.

### Context switch

The saving and restoring of computational state when switching between different threads or processes. In this manual, the term context switch describes any situation where the context is switched by an operating system and might or might not include changes to the address space.

### Context synchronization event

A context synchronization event is one of the following:

- Performing an ISB operation. An ISB operation is performed when an `ISB` instruction is executed and does not fail its condition code check.
- Taking an exception.
- Returning from an exception.
- Exit from Debug state.

For more information, see [B3.32 Context Synchronization Event on page 124](#).

#### Note

Security state transitions are not Context synchronization events.



### Control dependency

A control dependency exists when the data value that is returned by a read access determines the condition flags, and the values of the flags determine the address of a subsequent read access. This address determination might be through conditional execution, or through the evaluation of a branch.

### Cross Trigger Interface

A debug component that is not part of the Armv8-M architecture.

### CTI

See [Cross Trigger Interface](#).

### DAP

Debug Access Port.

### Data Watchpoint and Trace (DWT)

The Data Watchpoint and Trace unit is a component of Armv8-M debug that optionally provides a number of trace, sampling, and profiling functions.

See also [B12.2 Data Watchpoint and Trace unit on page 278](#).

### DCB

See [Debug Control Block](#).

### Debug Control Block (DCB)

A region in the System Control Space that is assigned to registers that support debug features.

See also [System Control Space](#).

### Debugger

In most of this manual, *debugger* refers to any agent that is performing debug. However, some parts of the manual require a more rigorous definition, and define debugger locally. See [Chapter B11 Debug on page 230](#).

### Deprecated

Something that is present in the Arm architecture for backwards compatibility. Whenever possible software must avoid using deprecated features. Features that are deprecated but are not optional are present in current implementations of the Arm architecture, but might not be present, or might be deprecated and **OPTIONAL**, in future versions of the Arm architecture.

See also **OPTIONAL**.

### Digital signal processing (DSP)

Algorithms for processing signals that have been sampled and converted to digital form. DSP algorithms often use saturated arithmetic.

### Direct access

A read or write of a register.

### Domain

In the Arm architecture, *domain* is used in the following contexts.

**Shareability domain** Defines a set of observers for which the Shareability attributes make the data or unified caches transparent for data accesses.

**Power domain** Defines a block of logic with a single, common, power supply.

### Double-precision value

Consists of two consecutive 32-bit words that are interpreted as a basic double-precision floating-point number according to the *IEEE Standard for Floating-point Arithmetic*.

**Doubleword**

A 64-bit data item. Doublewords are normally at least word-aligned in Arm systems.

**Doubleword-aligned**

Means that the address is divisible by 8.

**DSP**

See [Digital signal processing](#).

**DWT**

See [Data Watchpoint and Trace](#).

**Effective value**

A register control field, meaning a field in a register that controls some aspect of the behavior, can be described as having an *Effective value*:

- In some cases, the description of a particular control *a* specifies that when control *a* is active it causes a register control field *b* to be treated as having a fixed value for all purposes other than direct reads, or direct reads and direct writes, of the register containing control field *b*. When control *a* is active that fixed value is described as the *Effective value* of register control field *b*.

In other cases, a register control field *b* is not implemented or is not accessible, but behavior of the PE is as if control field *b* was implemented and accessible, and had a particular value. In this case, that value is the *Effective value* of register control field *b*.

Where a register control field is introduced in a particular version of the architecture, and is not implemented in an earlier version of the architecture, typically it will have an *Effective value* in that earlier version of the architecture.

- Otherwise, the *Effective value* of a register control field is the value of that field.

**Embedded Trace Macrocell (ETM)**

A component of the Arm CoreSight debug and trace solution. An ETM provides non-invasive trace of PE operation.

**Endianness**

An aspect of the system memory mapping. For more information, see [B5.3 Endianness on page 146](#).

See also [Big-endian memory](#) and [Little-endian memory](#).

**EPSR**

See [Execution Program Status Register](#).

**ETM**

See [Embedded Trace Macrocell](#)

**Exception**

Handles an event. For example, an exception could handle an external interrupt or an undefined instruction.

**Exception vector**

A fixed address that contains the address of the first instruction of the corresponding exception handler.

**Execution Program Status Register (EPSR)**

A register that contains the Execution state bits and is part of the XPSR.

See also [B3.5 XPSR, APSR, IPSR, and EPSR on page 59](#).

### Execution stream

The stream of instructions that would have been executed by sequential execution of the program.

### Explicit access

A read from memory, or a write to memory, generated by a load or store instruction that is executed by the PE.

### Flash Patch and Breakpoint Unit

The Flash Patch and Breakpoint unit supports setting breakpoints on instruction fetches.

See also [B12.5 Flash Patch and Breakpoint unit on page 301](#).

### Flush-to-zero mode

A processing mode that optimizes the performance of some floating-point algorithms by replacing the denormalized operands and Intermediate results with zeros, without significantly affecting the accuracy of their final results.

### FPB

See [Flash Patch and Breakpoint Unit](#).

### General-purpose registers

The registers that the base instructions use for processing:

- The general-purpose registers are R0-R12. R13-R14 are the SP and LR, respectively. For more information, see [B3.3 Registers on page 56](#).

See also [High registers](#), [Low registers](#).

### Halfword

A 16-bit data item. Halfwords are normally halfword-aligned in Arm systems.

### Halfword-aligned

Means that the address is divisible by 2.

### High registers

The general-purpose registers R8-R14. Most 16-bit T32 instructions cannot access the high registers.

#### Note

In some contexts, *high registers* refers to R8-R15, meaning R8-R14 and the PC.

See also [General-purpose registers](#), [Low registers](#).

### ICI

See [Interrupt continuable instruction](#).

### If-Then block (IT block)

An IT block is a block of up to four instructions following an *If-Then* (IT) instruction. Each instruction in the block is conditional. The conditions for the instructions are either all the same, or some are the inverse of others.

### Immediate and offset fields

Are unsigned unless otherwise stated.

### Immediate value

A value that is encoded directly in the instruction and used as numeric data when the instruction is executed. Many T32 instructions can be used with an immediate argument.

### IMP DEF

An abbreviation that is used in diagrams to indicate that one or more bits have IMPLEMENTATION DEFINED behavior.

### **IMPLEMENTATION DEFINED**

Means that the behavior is not architecturally defined, but must be defined and documented by individual implementations.

In body text, the term IMPLEMENTATION DEFINED is shown in SMALLCAPS.

### **Implicit access**

An access that is not explicit.

See also [Explicit access](#).

### **Imprecise exception**

An exception that is generated as the result of a system error. An imprecise exception is reported at the time that is asynchronous to the instruction that caused it.

### **Index register**

A register that is specified in some load and store instructions. The value of this register is used as an offset to be added to or subtracted from the base register value to form the address that is sent to memory. Some instruction forms permit the index register value to be shifted before the addition or subtraction.

### **Indirect access**

A read or write of a register that is not a direct access.

For example, an indirect write to a register might occur as the side-effect of executing an instruction that does not perform a direct write to the register, or because of some operation that is performed by an external agent.

See also [Direct access](#)

### **Inline literals**

These are constant addresses and other data items that are held in the same area as the software itself. They are automatically generated by compilers, and can also appear in assembler code.

### **Instrumentation Trace Macrocell (ITM)**

A component of the Arm CoreSight debug and trace solution. An ITM provides a memory-mapped register interface that applications can use to write logging or event words to a trace sink.

### **Interrupt continuable instruction**

Multicycle load or store instructions that can be interrupted part way through their execution. After the interrupt service routine has completed, execution of the partially executed instruction can be resumed and the instruction is not required to be restarted from the beginning. An interruption of a multicycle load or store instructions will cause the PE to set the [EPSR.ICI](#) fields.

### **Interrupt Program Status Register (IPSR)**

The register that provides status information on whether an application thread or exception handler is executing on the processor. If an exception handler is executing, the register provides information on the exception type. The register is part of the XPSR.

See also [B3.5 XPSR, APSR, IPSR, and EPSR on page 59](#).

### **Interrupt Service Routine**

The procedure that handles an interrupt.

### **Interworking**

A method of working that permits branches between software using the A32 and T32 instruction sets in the Armv8-A architecture. For Armv8-M, interworking is described in [C1.4.7 Instruction set, interworking and interstating support on page 323](#).

**IPSR**

See [Interrupt Program Status Register](#).

**ISR**

See [Interrupt Service Routine](#).

**ITM**

See [Instrumentation Trace Macrocell](#).

**Level**

See [Cache level](#).

**Level of Coherence (LoC)**

The last level of cache that must be cleaned or invalidated when cleaning or invalidating to the point of coherency.

See also [Cache level](#), [Point of Coherency](#).

**Level of Unification, Inner Shareable (LoUIS)**

The last level of cache that must be cleaned or invalidated when cleaning or invalidating to the point of unification for the Inner Shareable Shareability domain.

See also [Cache level](#), [Point of Unification](#).

**Level of Unification, uniprocessor (LoUU)**

For a PE, the last level of cache that must be cleaned or invalidated when cleaning or invalidating to the point of unification for that PE.

See also [Cache level](#), [Point of Unification](#).

**Line**

See [Cache line](#).

**Little-endian memory**

Means that, for example:

- A byte or halfword at a word-aligned address is the least significant byte or halfword in the word at that address.
- A byte at a halfword-aligned address is the least significant byte in the halfword at that address.

See also [Big-endian memory](#), [B5.3 Endianness on page 146](#).

**Load/store architecture**

An architecture where data-processing operations only operate on register contents, not directly on memory contents.

**LoC**

See [Level of Coherence](#).

**Lockup**

A PE state where the PE stops executing instructions in response to an error for which escalation to an appropriate HardFault handler is not possible because of the current execution priority. For more information, see [B3.31 Lockup on page 119](#).

**LoUIS**

See [Level of Unification, Inner Shareable](#).

## LoUU

See [Level of Unification, uniprocessor](#).

## Low registers

General-purpose registers R0-R7. Unlike the high registers, all T32 instructions can access the Low registers.

## Memory barriers

The term memory barrier is the general term that is applied to an instruction, or sequence of instructions, that forces synchronization events by a PE regarding retiring Load/Store instructions. For more information, see [B5.13 Memory barriers on page 161](#).

## Memory coherency

The problem of ensuring that when a memory location is read, either by a data read or an instruction fetch, the value that is actually obtained is always the value that was most recently written to the location. This can be difficult when there are multiple possible physical locations, such as main memory and at least one of a write buffer and one or more levels of cache.

## Memory hint

A memory hint instruction provides advance information to memory systems about future memory accesses, without actually loading or storing any data to or from the register file. PLD and PLI are the only memory hint instructions that are defined in Armv8-M.

## Memory Protection Unit (MPU)

A hardware unit whose registers provide simple control of a limited number of protection regions in memory, for more information, see [Chapter B8 The Armv8-M Protected Memory System Architecture on page 214](#).

## MPU

See [Chapter B8 The Armv8-M Protected Memory System Architecture on page 214](#).

## Multi-copy atomicity

The form of atomicity that is described in [B5.5.2 Multi-copy atomicity on page 149](#).

See also [Atomicity](#), [Single-copy atomicity](#).

## NaN

Not a Number. A floating-point value that can be used when neither a numeric value nor an infinity is appropriate. A NaN can be a *quiet* NaN, that propagate through most floating-point operations, or a *signaling* NaN, that causes an Invalid Operation floating-point exception when used. For more information, see the IEEE Standard for Floating-point Arithmetic.

## Non-Return-to-Zero (NRZ)

A physical layer signaling scheme that is used on asynchronous communication ports

## NRZ

See [Non-Return-to-Zero](#).

## Observer

A master in the system that is capable of observing memory accesses. For more information, see [B5.8 Observability of memory accesses on page 155](#).

## Obsolete

Obsolete indicates something that is no longer supported by Arm. When an architectural feature is described as obsolete, this indicates that the architecture has no support for that feature, although an earlier version of the architecture did support it.

### Offset addressing

Means that the memory address is formed by adding or subtracting an offset to or from the base register value.

### OPTIONAL

When applied to a feature of the architecture, OPTIONAL indicates a feature that is not required in an implementation of the Arm architecture:

- If a feature is OPTIONAL and deprecated, this indicates that the feature is being phased out of the architecture. Arm expects such a feature to be included in a new implementation only if there is a known backwards-compatibility reason for the inclusion of the feature.

A feature that is OPTIONAL and deprecated might not be present in future versions of the architecture.

- A feature that is OPTIONAL but not deprecated is, typically, a feature added to a version of the Arm architecture after the initial release of that version of the architecture. Arm recommends that such features are included in all new implementations of the architecture.

In body text, these meanings of the term OPTIONAL are shown in SMALLCAPS.

Note: Do not confuse these Arm-specific uses of OPTIONAL with other uses of OPTIONAL, where it has its usual meaning. These include:

- Optional arguments in the syntax of many instructions.
- Behavior that is determined by an implementation choice.

See also [Deprecated](#).

### PE

See [Processing element](#).

### Physical address (PA)

An address that identifies a location in the physical memory map.

### PoC

See [Point of Coherency](#).

### Point of coherency (PoC)

For a particular PA, the point at which all agents that can access memory are guaranteed to see the same copy of a memory location.

### Point of unification (PoU)

For a particular PE, the point by which the instruction and data caches of that PE are guaranteed to see the same copy of a memory location.

### Post-indexed addressing

Means that the memory address is the base register value, but an offset is added to or subtracted from the base register value and the result is written back to the base register.

### PoU

See [Point of Unification](#).

### PPB

Private Peripheral Bus

### Pre-indexed addressing

Means that the memory address is formed in the same way as for offset addressing, but the memory address is also written back to the base register.

### Prefetching

Prefetching refers to speculatively fetching instructions or data from the memory system. In particular, instruction prefetching is the process of fetching instructions from memory before the instructions that precede them, in simple sequential execution of the program, have finished executing. Prefetching an instruction does not mean that the instruction has to be executed.

In this manual, references to instruction or data fetching apply also to prefetching, unless the context explicitly indicates otherwise.

See also [Simple sequential execution](#).

### Privileged access

Memory systems typically differentiate between privileged and unprivileged accesses, and support more restrictive permissions for unprivileged accesses. Some instructions can be used only by privileged software.

### Processing element (PE)

The abstract machine that is defined in the Arm architecture, as documented in an Arm Architecture Reference Manual. A PE implementation compliant with the Arm architecture must conform with the behaviors described in the corresponding Arm Architecture Reference Manual.

### Program Status Registers (XPSR)

XPSR is the term that is used to describe the combination of the APSR, EPSR, and IPSR into a single 32-bit Program Status Register.

See also [B3.5 XPSR, APSR, IPSR, and EPSR on page 59](#).

### Protection region

A memory region whose position, size, and other properties are defined by Memory Protection Unit registers.

### Protection Unit

See [Memory Protection Unit](#)

### Pseudo-instruction

UAL assembler syntax that assembles to an instruction encoding that is expected to disassemble to a different assembler syntax, and is described in this manual under that other syntax. For example, `MOV<Rd>, <Rm>, LSL #<n>` is a pseudo-instruction that is expected to disassemble as `LSL<Rd>, <Rm>, #<n>`.

See also [Chapter C1 Instruction Set Overview on page 305](#).

### Quadword

A 128-bit data item. Quadwords are normally at least word-aligned in Arm systems.

### Quadword-aligned

Means that the address is divisible by 16.

### Quiet NaN

A NaN that propagates unchanged through most floating-point operations.

### RAO

See [Read-As-One](#).

### RAO/SBOP



In versions of the Arm architecture before Armv8, Read-As-One, Should-Be-One-or-Preserved on writes. In Armv8, RES1 replaces this description.

See also [UNK/SBOP](#), [Read-As-One](#), [RES1](#), [Should-Be-One-or-Preserved \(SBOP\)](#).

## RAO/WI

Read-As-One, Writes Ignored.

Hardware must implement the field as Read-As-One, and must ignore writes to the field.

Software can rely on the field reading as all 1s, and on writes being ignored.

This description can apply to a single bit that reads as 1, or to a field that reads as all 1s.

See also [Read-As-One](#).

## RAZ

See [Read-As-Zero](#).

## RAZ/SBZP

In versions of the Arm architecture before Armv8, Read-As-Zero, Should-Be-Zero-or-Preserved on writes. In Armv8, RES0 replaces this description.

See also [UNK/SBZP](#), [Read-As-Zero](#), [RES0](#), [Should-Be-Zero-or-Preserved \(SBOP\)](#).

## RAZ/WI

Read-As-Zero, Writes Ignored.

Hardware must implement the field as Read-As-Zero, and must ignore writes to the field.

Software can rely on the field reading as all 0s, and on writes being ignored.

This description can apply to a single bit that reads as 0, or to a field that reads as all 0s.

See also [Read-As-Zero](#).

## Read, modify, write

In a read, modify, write instruction sequence, a value is read to a general-purpose register, the relevant fields that are updated in that register, and the new value that is written back.

## Read-allocate cache

A cache in which a cache miss on reading data causes a cache line to be allocated into the cache.

## Read-As-One (RAO)

Hardware must implement the field as reading as all 1s.

Software:

- Can rely on the field reading as all 1s.
- Must use a [SBOP](#) policy to write to the field.

This description can apply to a single bit that reads as 1, or to a field that reads as all 1s. It applies only to a bit or field that is read-only.

See also [RAO/SBOP](#), [RAO/WI](#), [RES1](#).

## Read-As-Zero (RAZ)

Hardware must implement the field as reading as all 0s.

Software:

- Can rely on the field reading as all 0s

- Must use a [SBOP](#) policy to write to the field.

This description can apply to a single bit that reads as 0, or to a field that reads as all 0s. It applies only to a bit or field that is read-only.

See also [RAZ/SBZP](#), [RAZ/WI](#), [RES0](#).

### Register data dependency

A register data dependency exists between a first data value and a second data value when either:

- The register that holds the first data value is used in the calculation of the second data value, and the calculation between the first data value and the second data value does not consist of either:
  - A conditional branch whose condition is determined by the first data value.
  - A conditional selection, move, or computation whose condition is determined by the first data value, where the input data values for the selection, move, or computation do not have a data dependency on the first data value.
- There is a register data dependency between the first data value and a third data value, and between the third data value and the second data value.

### RES0

A reserved bit or field with [Should-Be-Zero-or-Preserved](#) behavior, or equivalent read-only or write-only behavior. Used for fields in register descriptions, and for fields in architecturally defined data structures that are held in memory.

Within the architecture, there are some cases where a register bit or field:

- Is RES0 in some defined architectural context.
- Has different defined behavior in a different architectural context.

#### Note

RES0 is not used in descriptions of instruction encodings.

This means the definition of RES0 for fields in read/write registers is:

#### If a bit is RES0 in all contexts

For a bit in a read/write register, it is IMPLEMENTATION DEFINED whether:

1. The bit is hardwired to 0. In this case:
  - Reads of the bit always return 0.
  - Writes to the bit are ignored.
2. The bit can be written. In this case:
  - An indirect write to the register sets the bit to 0.
  - A read of the bit returns the last value that is successfully written, by either a direct or an indirect write, to the bit.

If the bit has not been successfully written since reset, then the read of the bit returns the reset value if there is one, or otherwise returns an UNKNOWN value.

- A direct write to the bit must update a storage location that is associated with the bit.
- The value of the bit must have no effect on the operation of the PE, other than determining the value read back from the bit, unless this manual explicitly defines additional properties for the bit.

Whether RES0 bits or fields follow behavior 1 or behavior 2 is IMPLEMENTATION DEFINED on a field-by-field basis.

#### If a bit is RES0 only in some contexts

For a bit in a read/write register, when the bit is described as RES0:

- An indirect write to the register sets the bit to 0.
- A read of the bit must return the value last successfully written to the bit, by either a direct or an indirect write, regardless of the use of the register when the bit was written.  
If the bit has not been successfully written since reset, then the read of the bit returns the reset value if there is one, or otherwise returns an UNKNOWN value.
- A direct write to the bit must update a storage location that is associated with the bit.
- While the use of the register is such that the bit is described as RES0, the value of the bit must have no effect on the operation of the PE, other than determining the value read back from that bit, unless this manual explicitly defines additional properties for the bit.

Considering only contexts that apply to a particular implementation, if there is a context in which a bit is defined as RES0, another context in which the same bit is defined as RES1, and no context in which the bit is defined as a functional bit, then it is IMPLEMENTATION DEFINED whether:

- Writes to the bit are ignored, and reads of the bit return an UNKNOWN value.
- The value of the bit can be written, and a read returns the last value that is written to the bit.

The RES0 description can apply to bits or fields that are read-only, or are write-only:

- For a read-only bit, RES0 indicates that the bit reads as 0, but software must treat the bit as UNKNOWN.
- For a write-only bit, RES0 indicates that software must treat the bit as SBZ.

A bit that is RES0 in a context is reserved for possible future use in that context. To preserve forward compatibility, software:

- Must not rely on the bit reading as 0.
- Must use an policy to write to the bit.

This RES0 description can apply to a single bit, or to a field for which each bit of the field must be treated as RES0.

In body text, the term RES0 is shown in SMALLCAPS.

See also [Read-As-Zero](#), [RES1](#), [Should-Be-Zero-or-Preserved](#), [UNKNOWN](#).

## RES0H

A reserved bit or field with [Should-Be-Zero-or-Preserved \(SBZP\)](#). This behavior uses the *Hardwired to 0* subset of the RES0 definition.

## RES1

A reserved bit or field with [Should-Be-One-or-Preserved](#) behavior, or equivalent read-only or write-only behavior. Used for fields in register descriptions, and for fields in architecturally defined data structures that are held in memory.

Within the architecture, there are some cases where a register bit or field:

- Is RES1 in some defined architectural context.
- Has different defined behavior in a different architectural context.

### Note

RES1 is not used in descriptions of instruction encodings.

This means the definition of RES1 for fields in read/write registers is:

### If a bit is RES1 in all contexts

For a bit in a read/write register, it is IMPLEMENTATION DEFINED whether:

1. The bit is hardwired to 1. In this case:
  - Reads of the bit always return 1.
  - Writes to the bit are ignored.
2. The bit can be written. In this case:

- An indirect write to the register sets the bit to 1.
- A read of the bit returns the last value that is successfully written, by either a direct or an indirect write, to the bit.  
If the bit has not been successfully written since reset, then the read of the bit returns the reset value if there is one, or otherwise returns an UNKNOWN value.
- A direct write to the bit must update a storage location that is associated with the bit.
- The value of the bit must have no effect on the operation of the PE, other than determining the value read back from the bit, unless this manual explicitly defines additional properties for the bit.

Whether RES1 bits or fields follow behavior 1 or behavior 2 is IMPLEMENTATION DEFINED on a field-by-field basis.

#### **If a bit is RES1 only in some contexts**

For a bit in a read/write register, when the bit is described as RES1:

- An indirect write to the register sets the bit to 1.
- A read of the bit must return the value last successfully written to the bit, regardless of the use of the register when the bit was written.

#### **Note**

As indicated in this list, this value might be written by an indirect write to the register.

If the bit has not been successfully written since reset, then the read of the bit returns the reset value if there is one, or otherwise returns an UNKNOWN value.

- A direct write to the bit must update a storage location that is associated with the bit.
- While the use of the register is such that the bit is described as RES1, the value of the bit must have no effect on the operation of the PE, other than determining the value read back from that bit, unless this manual explicitly defines additional properties for the bit.

Considering only contexts that apply to a particular implementation, if there is a context in which a bit is defined as RES0, another context in which the same bit is defined as RES1, and no context in which the bit is defined as a functional bit, then it is IMPLEMENTATION DEFINED whether:

- Writes to the bit are ignored, and reads of the bit return an UNKNOWN value.
- The value of the bit can be written, and a read returns the last value that is written to the bit.

The RES1 description can apply to bits or fields that are read-only, or are write-only:

- For a read-only bit, RES1 indicates that the bit reads as 1, but software must treat the bit as UNKNOWN.
- For a write-only bit, RES1 indicates that software must treat the bit as [SBO](#).

A bit that is RES1 in a context is reserved for possible future use in that context. To preserve forward compatibility, software:

- Must not rely on the bit reading as 1.
- Must use an [SBOP](#) policy to write to the bit.

This RES1 description can apply to a single bit, or to a field for which each bit of the field must be treated as RES1.

In body text, the term RES1 is shown in SMALLCAPS.

See also [Read-As-One](#), [RES0](#), [Should-Be-One-or-Preserved](#), [UNKNOWN](#).

## **RES1H**

A reserved bit or field with [Should-Be-One-or-Preserved \(SBOP\)](#) behavior. This behavior uses the *Hardwired to 1* subset of the [RES1](#) definition.

## **Reserved**

Unless otherwise stated:

- Instructions that are reserved or that access reserved registers have UNPREDICTABLE or CONSTRAINED UNPREDICTABLE behavior.
- Bit positions that are described as reserved are:
  - In an RW or WO register, RES0.
  - In an RO register, UNK.

See also [CONSTRAINED UNPREDICTABLE](#), [RES0](#), [RES1](#), [UNDEFINED](#), [UNK](#), [UNPREDICTABLE](#).

### Return Link

A value relating to the return address.

### RISC

Reduced Instruction Set Computer.

### Rounding error

The value of the rounded result of an arithmetic operation minus the exact result of the operation.

### Rounding mode

Specifies how the exact result of a floating-point operation is rounded to a value that is representable in the destination format. The rounding modes are defined by the *IEEE Standard for Floating-point Arithmetic*.

### Saturated arithmetic

Integer arithmetic in which a result that would be greater than the largest representable number is set to the largest representable number, and a result that would be less than the smallest representable number is set to the smallest representable number. Signed saturated arithmetic is often used in DSP algorithms. It contrasts with the normal signed integer arithmetic used in Arm processors, in which overflowing results wrap around from  $+2^{31} - 1$  to  $-2^{31}$  or the opposite way.

### SBO

See [Should-Be-One](#).

### SBOP

See [Should-Be-One-or-Preserved](#).

### SBZ

See [Should-Be-Zero](#).

### SBZP

See [Should-Be-Zero-or-Preserved](#)

### Security hole

A mechanism by which execution at the current level of privilege can achieve an outcome that cannot be achieved at the current or a lower level of privilege using instructions that are not UNPREDICTABLE and are not CONSTRAINED UNPREDICTABLE. The Arm architecture forbids security holes.

See also [CONSTRAINED UNPREDICTABLE](#), [UNPREDICTABLE](#).

### Self-modifying code

Code that writes one or more instructions to memory and then executes them. When using self-modifying code, cache maintenance and barrier instructions must be used to ensure synchronization.

### Serial Wire Output (SWO)

An asynchronous TPIU port supporting one or both of the NRZ and Manchester encodings.

### Serial Wire Viewer (SWV)

The combination of an SWO and at least one of a DWT unit or an ITM, providing data tracing capability.

### Set

See [Cache sets](#).

### Should-Be-One (SBO)

Hardware must ignore writes to the field.

Arm strongly recommends that software writes the field as all 1s. If software writes a value that is not all 1s, it must expect an UNPREDICTABLE or CONSTRAINED UNPREDICTABLE result.

This description can apply to a single bit that should be written as 1, or to a field that should be written as all 1s.

See also [CONSTRAINED UNPREDICTABLE](#), [UNPREDICTABLE](#).

### Should-Be-One-or-Preserved (SBOP)

From the introduction of the Armv8 architecture, the description *Should-Be-One-or-Preserved* is superseded by RES1.

Hardware must ignore writes to the field.

If software has read the field since the PE implementing the field was last reset and initialized, it must preserve the value of the field by writing the value that it previously read from the field. Otherwise, it must write the field as all 1s.

If software writes a value to the field that is not a value that is previously read for the field and is not all 1s, it must expect an UNPREDICTABLE or CONSTRAINED UNPREDICTABLE result.

This description can apply to a single bit that should be written as its preserved value or as 1, or to a field that should be written as its preserved value or as all 1s.

See also [CONSTRAINED UNPREDICTABLE](#), [UNPREDICTABLE](#).

### Should-Be-Zero (SBZ)

Hardware must ignore writes to the field.

Arm strongly recommends that software writes the field as all 0s. If software writes a value that is not all 0s, it must expect an UNPREDICTABLE or CONSTRAINED UNPREDICTABLE result.

This description can apply to a single bit that should be written as 0, or to a field that should be written as all 0s.

See also [CONSTRAINED UNPREDICTABLE](#), [UNPREDICTABLE](#).

### Should-Be-Zero-or-Preserved (SBZP)

From the introduction of the Armv8 architecture, the description *Should-Be-Zero-or-Preserved* is superseded by RES0.

Hardware must ignore writes to the field.

If software has read the field since the PE implementing the field was last reset and initialized, it must preserve the value of the field by writing the value that it previously read from the field. Otherwise, it must write the field as all 0s.

If software writes a value to the field that is not a value that is previously read for the field and is not all 0s, it must expect an UNPREDICTABLE or CONSTRAINED UNPREDICTABLE result.

This description can apply to a single bit that should be written as its preserved value or as 0, or to a field that should be written as its preserved value or as all 0s.

See also [CONSTRAINED UNPREDICTABLE](#), [UNPREDICTABLE](#).

### Signaling NaNs

Cause an Invalid Operation exception whenever any floating-point operation receives a signaling NaN as an operand. Signaling NaNs can be used in debugging, to track down some uses of uninitialized variables.

### Signed data types

Represent an integer in the range  $-2^{N-1}$  to  $+2^{N-1} - 1$ , using two's complement format.

### Signed immediate and offset fields

Are encoded in two's complement notation unless otherwise stated.

### SIMD

Single-Instruction, Multiple-Data.

### Simple sequential execution

The behavior of an implementation that fetches, decodes and completely executes each instruction before proceeding to the next instruction. Such an implementation performs no Speculative accesses to memory, including to instruction memory. The implementation does not pipeline any phase of execution. In practice, this is the theoretical execution model that the architecture is based on, and Arm does not expect this model to correspond to a realistic implementation of the architecture.

### Single peripheral

A single peripheral is a region of memory of an IMPLEMENTATION DEFINED size that is defined by the peripheral.

### Single-copy atomicity

The form of atomicity that is described in [B5.5.1 Single-copy atomicity on page 149](#).

See also [Atomicity](#), [Multi-copy atomicity](#).

### Single-precision value

A 32-bit word that is interpreted as a basic single-precision floating-point number according to the IEEE Standard for Floating-point Arithmetic.

### Spatial locality

The observed effect that after a program has accessed a memory location, it is likely to also access nearby memory locations in the near future. Caches with multi-word cache lines exploit this effect to improve performance.

### Special-purpose register

One of a specified set of registers for which all direct and indirect reads and writes to the register appear to occur in program order relative to other instructions, without the need for any explicit synchronization. For more information, see [B3.3 Registers on page 56](#).

### Speculative writes

All of the following are Speculative writes:

- Writes generated by store instructions that appear in the Execution stream after a branch that is not architecturally resolved.
- Writes generated by store instructions that appear in the Execution stream after an instruction where a synchronous exception condition has not been architecturally resolved.
- Writes generated by conditional store instructions for which the conditions for the instruction have not been architecturally resolved.
- Writes generated by store instructions for which the data being written comes from a register that has not been architecturally committed.

### System Control Block (SCB)

An address region in the System Control Space, which is used for key feature control and configuration that is associated with the exception model.

See also [System Control Space](#).

### **System Control Space (SCS)**

A region of the memory map that is reserved for system control and configuration registers.

See also [Debug Control Block](#), [B6.3 The System Control Space \(SCS\)](#) on page 202.

### **T32 instruction**

One or two halfwords that specify an operation to be performed by a PE. T32 instructions must be halfword-aligned. For more information, see [Chapter C1 Instruction Set Overview](#) on page 305.

T32 instructions were previously called Thumb instructions.

### **Tail-chaining**

An optimization that removes unstacking and stacking operations. For more information, see [B3.26 Tail-chaining](#) on page 108.

### **Temporal locality**

The observed effect that after a program has accesses a memory location, it is likely to access the same memory location again in the near future. Caches exploit this effect to improve performance.

### **TPIU**

See [Trace Port Interface Unit](#).

### **Trace Port Interface Unit (TPIU)**

A component of the Arm CoreSight debug and trace solution. A TPIU provides an external interface for one or more trace sources in the processor implementation.

### **UAL**

See [Unified Assembler Language](#).

### **Unaligned**

An unaligned access is an access where the address of the access is not aligned to the size of an element of the access.

### **Unaligned memory accesses**

Are memory accesses that are not, or might not be, appropriately halfword-aligned, word-aligned, or doubleword-aligned.

### **Unallocated**

Except where otherwise stated in this manual, an instruction encoding is unallocated if the architecture does not assign a specific function to the entire bit pattern of the instruction, but instead describes it as `CONSTRAINED UNPREDICTABLE`, `UNDEFINED`, `UNPREDICTABLE`, or as an unallocated hint instruction.

A bit in a register is unallocated if the architecture does not assign a function to that bit.

See also [CONSTRAINED UNPREDICTABLE](#), [UNPREDICTABLE](#), [UNDEFINED](#).

### **UNDEFINED**

Indicates an instruction that generates an Undefined Instruction exception.

In body text, the term `UNDEFINED` is shown in SMALLCAPS.

See also [Chapter C1 Instruction Set Overview](#) on page 305.



## Unified Assembler Language

The assembler language that is introduced with Thumb-2 technology that is used in this manual. See [Chapter C1 Instruction Set Overview](#) on page 305 for details.

## Unified cache

Is a cache that is used for both processing instruction fetches and processing data loads and stores.

## Unindexed addressing

Means addressing in which the base register value is used directly as the address to send to memory, without adding or subtracting an offset. In most types of load/store instruction, unindexed addressing is performed by using offset addressing with an immediate offset of 0.

In the M-Profile, the LDC, LDC2, STC, and STC2 instructions have an explicit unindexed addressing mode that permits the offset field in the instruction to specify additional coprocessor options.

## UNK

An abbreviation indicating that software must treat a field as containing an UNKNOWN value.

Hardware must implement the bit as read as 0, or all 0s for a multi-bit field. Software must not rely on the field reading as zero.

See also [UNKNOWN](#).

## UNK/SBOP

Hardware must implement the field as Read-As-One, and must ignore writes to the field.

Software must not rely on the field reading as all 1s, and except for writing back to the register it must treat the value as if it is UNKNOWN. Software must use an SBOP policy to write to the field.

This description can apply to a single bit that should be written as its preserved value or as 1, or to a field that should be written as its preserved value or as all 1s.

See also [Read-as-One](#), [Should-Be-One-or-Preserved](#), [UNKNOWN](#).

## UNK/SBZP

Hardware must implement the bit as Read-As-Zero, and must ignore writes to the field.

Software must not rely on the field reading as all 0s, and except for writing back to the register must treat the value as if it is UNKNOWN. Software must use an SBZP policy to write to the field.

This description can apply to a single bit that should be written as its preserved value or as 0, or to a field that should be written as its preserved value or as all 0s.

See also [Read-as-Zero](#), [Should-Be-Zero-or-Preserved](#), [UNKNOWN](#).

## UNKNOWN

An UNKNOWN value does not contain valid data, and can vary from moment to moment, instruction to instruction, and implementation to implementation. An UNKNOWN value must not return information that cannot be accessed at the current or a lower level of privilege using functionality that is not UNKNOWN, is not CONSTRAINED UNPREDICTABLE, and does not return UNKNOWN values.

An Unknown value must not be documented or promoted as having a defined value or effect.

In body text, the term UNKNOWN is shown in SMALLCAPS.

See also [CONSTRAINED UNPREDICTABLE](#), [UNDEFINED](#), [UNK](#), [UNPREDICTABLE](#).

## UNPREDICTABLE

## Glossary

Means the behavior cannot be relied on. UNPREDICTABLE behavior must not perform any function that cannot be performed at the current or a lower level of privilege or security using instructions that are not UNPREDICTABLE.

UNPREDICTABLE behavior must not be documented or promoted as having a defined effect.

An instruction that is UNPREDICTABLE can be implemented as UNDEFINED.

In body text, the term UNPREDICTABLE is shown in SMALLCAPS.

See also [CONSTRAINED UNPREDICTABLE](#), [UNDEFINED](#).

### Unsigned data types

Represent a non-negative integer in the range 0 to  $+2^{N-1} - 1$ , using normal binary format.

### Watchpoint

A debug event that is triggered by an access to memory, which is specified in terms of the address of the location in memory being accessed.

### Way

See [Cache way](#).

### WI

Writes Ignored. In a register that software can write to, a WI attribute that is applied to a bit or field indicates that the bit or field ignores the value that is written by software and retains the value it had before that write.

See also [RAO/WI](#), [RAZ/WI](#), [RES0](#), [RES1](#).

### Word

A 32-bit data item. Words are normally word-aligned in Arm systems.

### Word-aligned

Means that the address is divisible by 4.

### Write buffer

A block of high-speed memory that optimizes stores to main memory.

### Write-Allocate cache

A cache in which a cache miss on storing data causes a cache line to be allocated into the cache.

### Write-back cache

A cache in which when a cache hit occurs on a store access, the data is only written to the cache. Data in the cache can therefore be more up-to-date than data in main memory. Any such data is written back to main memory when the cache line is cleaned or reallocated. Another common term for a write-back cache is a *copy-back cache*.

### Write-one-to-clear

Writing 1 to the relevant bit clears it to 0. Writing 0 to the bit has no effect.

### Write-one-to-set

Writing 1 to the relevant bit sets it to 0. Writing 0 to the bit has no effect.

### Write-Through cache

A cache in which when a cache hit occurs on a store access, the data is written both to the cache and to main memory. This is normally done using a write buffer, to avoid slowing down the PE.

### XPSR

See [Program Status Registers \(XPSR\)](#)