ARM Architecture Reference Manual

Disclaimer:

This document is provided as a Confidential draft of work in progress. There is known, ongoing work in several areas. Please see the associated release notes for details.

In addition, an exhaustive check of "TBDs" is not possible prior to this release, and as such, apologies for any that exist due to the extensive text changes. The majority of TBDs should be cross-reference related.

Confidential - Draft

ARM®

ARM Architecture Reference Manual

Copyright © 1996–2003 ARM Limited. All rights reserved.

Release Information

The following changes have been made to this document.

Change History

Date	Issue	Change
February 1996	A	First edition.
July 1997	B	Updated and index added.
April 1998	C	Updated.
February 2000	D	Updated for ARM architecture v5.
June 2000	E	Updated for ARM architecture v5TE and corrections to Part B.
July 2003	F-02	First general review

Proprietary Notice

ARM, the ARM Powered logo, Thumb, and StrongARM are registered trademarks of ARM Limited.

The ARM logo, AMBA, Angel, ARMulator, EmbeddedICE, ModelGen, Multi-ICE, PrimeCell, ARM7TDMI, ARM7TDMI-S, ARM9TDMI, ARM9E-S, ETM7, ETM9, TDMI, STRONG, are trademarks of ARM Limited.

All other products or services mentioned herein may be trademarks of their respective owners.

Neither the whole nor any part of the information contained in, or the product described in, this document may be adapted or reproduced in any material form except with the prior written permission of the copyright holder.

The product described in this document is subject to continuous developments and improvements. All particulars of the product and its use contained in this document are given by ARM in good faith. However, all warranties implied or expressed, including but not limited to implied warranties of merchantability, or fitness for purpose, are excluded.

This document is intended only to assist the reader in the use of the product. ARM Limited shall not be liable for any loss or damage arising from the use of any information in this document, or any error or omission in such information, or any incorrect use of the product.

Contents
ARM Architecture Reference Manual

Chapter A3   The ARM Instruction Set

Chapter A4   ARM Instructions

Chapter A5   ARM Addressing Modes

Chapter A6   The Thumb Instruction Set

Chapter A7   Thumb Instructions

Part B   Memory and System Architectures

Chapter B1   Introduction to Memory and System Architectures

Chapter B2   Memory Order Model

Chapter B3   The System Control Coprocessor

Chapter B4   Virtual Memory System Architecture

Chapter B5   Protected Memory System Architecture

Part D   Debug Architecture

Preface

This preface describes the versions of the ARM architecture and the contents of this manual, then lists the conventions and terminology it uses.

- *About this manual* on page x
- *Architecture versions and variants* on page xi
- *Using this manual* on page xvii
- *Conventions* on page xix.

About this manual

The purpose of this manual is to describe the ARM instruction set architecture, including its high code density Thumb subset, and two of its standard coprocessor extensions:

• The standard System Control coprocessor (coprocessor 15), which is used to control memory system components such as caches, write buffers, Memory Management Units, and Protection Units.

• The *Vector Floating-point* (VFP) architecture, which uses coprocessors 10 and 11 to supply a high-performance floating-point instruction set.

These instruction sets are described primarily from the viewpoint of the instruction being a 32-bit word or 16-bit halfword. The precise effects of each instruction are described, including any restrictions on its use. This information is of primary importance to authors of compilers, assemblers, and other programs that generate ARM machine code.

Assembler syntax is given for most of the instructions described in this manual, allowing instructions to be specified in textual form. This is of considerable use to assembly code writers, and also when debugging either assembler or high-level language code at the single instruction level.

However, this manual is not intended as tutorial material for ARM assembler language, nor does it describe ARM assembler language at anything other than a very basic level. To make effective use of ARM assembler language, consult the documentation supplied with the assembler being used. Different assemblers vary considerably with respect to many aspects of assembler language, such as which assembler directives are accepted and how they are coded.

A considerable amount of generic information is also included about how ARM processors access memory and other system components. Although this usually needs to be supplemented by detailed implementation-specific information from the technical reference manual of the device being used, this information is of use to designers of ARM-based systems.

Architecture versions and variants

The ARM instruction set architecture has evolved significantly since it was first developed, and will continue to be developed in the future. To be precise about which instructions exist in any particular ARM implementation, five major versions of the instruction set have been defined to date. These are denoted by the version numbers 1 to 5.

Many of the versions can be qualified with *variant* letters to specify collections of additional instructions that are included in that version. These collections vary from being very small (the M variant denotes the addition of just four extra instructions) to very large (the T variant denotes the addition of the entire Thumb instruction set).

The five versions of the ARM instruction set architecture to date are as follows:

Version 1   This version was implemented only by ARM1, and was never used in a commercial product. It contained:

- the basic data-processing instructions (not including multiplies)
- byte, word, and multi-word load/store instructions
- branch instructions, including a branch-and-link instruction designed for subroutine calls
- a software interrupt instruction, for use in making Operating System calls.

Version 1 only had a 26-bit address space, and is now obsolete.

Version 2   This version extended architecture version 1 by adding:

- multiply and multiply-accumulate instructions
- coprocessor support
- two more banked registers in fast interrupt mode
- atomic load-and-store instructions called SWP and SWPB (in a slightly later variant called version 2a).

Versions 2 and 2a still only had a 26-bit address space, and are now obsolete.

Version 3   This architecture version extended the addressing range to 32 bits. Program status information which had previously been stored in R15 was moved to a new *Current Program Status Register* (CPSR), and *Saved Program Status Registers* (SPSRs) were added to preserve the CPSR contents when an exception occurred. As a result, the following changes occurred to the instruction set:

- two instructions (MRS and MSR) were added to allow the new CPSR and SPSRs to be accessed.
- the functionality of instructions previously used to return from exceptions was modified to allow them to continue to be used for that purpose.

Version 3 also added two new processor modes to make it possible to use Data Abort, Prefetch Abort and Undefined Instruction exceptions effectively in Operating System code.

Backwards-compatibility support for the 26-bit architectures was obligatory in version 3, except in a variant called version 3G. The distinction between versions 3 and 3G is now obsolete.

Version 4   This version extended architecture version 3 by adding:

- halfword load/store instructions
- instructions to load and sign-extend bytes and halfwords
- in T variants, an instruction to transfer to Thumb state
- a new privileged processor mode that uses the User mode registers.

Version 4 also made it clearer which instructions should cause the Undefined Instruction exception to be taken.

Backwards-compatibility support for 26-bit architectures ceased to be obligatory in version 4.

Version 5   This version extended architecture version 4 by adding instructions and slightly modifying the definitions of some existing instructions to:

- improve the efficiency of ARM/Thumb interworking in T variants
- allow the same code generation techniques to be used for non-T variants as for T variants.

Version 5 also:

- added a *count leading zeros* instruction, which (among other things) allows more efficient integer divide and interrupt prioritization routines
- added a software breakpoint instruction
- added more instruction options for coprocessor designers
- tightens the definition of how flags are set by multiply instructions.

Version 6   This version extends architecture version 5 by adding instructions and slightly modifying the definitions of some existing instructions to:

- tbd
- tbd.

Version 6 also:

- tbd
- adds tbd instruction
- adds tbd
- tightens tbd.

The Thumb instruction set (T variants)

The Thumb instruction set is a re-encoded subset of the ARM instruction set. Thumb instructions are half the size of ARM instructions (16 bits compared with 32), with the result that greater code density can usually be achieved by using the Thumb instruction set instead of the ARM instruction set. The Thumb instruction set is described in detail in Chapter A6 *The Thumb Instruction Set* and Chapter A7 *Thumb Instructions*.

Two limitations of the Thumb instruction set compared with the ARM instruction set are:

- Thumb code usually uses more instructions for the same job, so ARM code is usually best for maximizing the performance of time-critical code.

ARM DDI 0100F-02

- The Thumb instruction set does not include some instructions that are needed for exception handling, so ARM code needs to be used for at least the top-level exception handlers.

Because of the second of these, the Thumb instruction set is always used in conjunction with a suitable version of the ARM instruction set. Its presence is denoted by the variant letter T, and it is not valid prior to ARM architecture version 4.

Thumb instruction set versions

There are two versions of the Thumb instruction set:

•   Thumb version 1 is used in T variants of ARM architecture version 4

•   Thumb version 2 is used in T variants of ARM architecture version 5.

Compared with Thumb version 1, Thumb version 2:

•   adds instructions and slightly modifies the definition of some existing instructions to improve the efficiency of ARM/Thumb interworking

•   adds a software breakpoint instruction

•   tightens the definition of how the Thumb multiply instruction sets the flags.

These improvements are closely related to the changes between ARM architecture versions 4 and 5.

——— **Note** ———

In general, the Thumb instruction set version number is not used in this manual. Instead, the version number of the associated version of the ARM instruction set is used, to allow easy use with the naming scheme described in *Naming of ARM/Thumb architecture versions* on page xv.

Long multiply instructions (M variants)

M variants of the ARM instruction set include four extra instructions which perform $32 \times 32 \rightarrow 64$ multiplications and $32 \times 32 + 64 \rightarrow 64$ multiply-accumulates. These instructions imply the existence of a multiplier that is significantly larger than minimum, and so are sometimes omitted in implementations for which a small die size is very important and multiply performance is not very important. Their presence is denoted by the use of the variant letter M.

These instructions were first defined as a variant of architecture version 3, and are included in similar variants of later architecture versions. Because the combination of requirements that leads to them being excluded does not arise very often in practice, inclusion of these instructions is standard in architecture versions 4 and above.

Enhanced DSP instructions (E variants)

E variants of the ARM instruction set include a number of extra instructions which enhance the performance of an ARM processor on typical *digital signal processing* (DSP) algorithms. These instructions are described in detail in Chapter A4 *ARM Instructions*, and include:

•   Several multiply and multiply-accumulate instructions that act on 16-bit data items

•   Addition and subtraction instructions that perform *saturated* signed arithmetic. This is a form of integer arithmetic that produces the maximum negative or positive value instead of wrapping around if the calculation overflows the normal integer range.

- Load (LDRD), store (STRD) and coprocessor register transfer (MCRR and MRRC) instructions that act on 2 words of data.

- A cache preload instruction PLD.

These instructions were first defined as a variant of architecture version 5T. Their presence is denoted by the variant letter E, and they are not valid prior to architecture version 5. They are also not valid in non-T or non-M variants of the architecture.

The ARMv5TExP architecture version

Some early implementations of the enhanced DSP variant of the ARM architecture omitted the LDRD, STRD, MCRR, MRRC and PLD instructions. Apart from this omission, all the ARM implementations concerned implemented the ARMv5TE architecture.

To be able to name this architecture variant, the letter P can be used to exclude these five instructions from architecture version ARMv5TE, according to the rules in *Naming of ARM/Thumb architecture versions*. The resulting architecture variant is therefore named ARMv5TExP. This is the only use of the P variant letter.

Naming of ARM/Thumb architecture versions

To name a precise version and variant of the ARM/Thumb architecture, the following strings are concatenated:

1. The string ARMv.
2. The version number of the ARM instruction set.
3. Variant letters of the included variants, except that the M variant is standard in architecture versions 4 and above, and therefore not normally listed.
4. If any variants described as *standard* in 3 above are not present, the letter x followed by the letters of the excluded variants. In addition, the letter P can be used after x to denote the exclusion of certain instructions from architecture version ARMv5TE, as described in *The ARMv5TExP architecture version*.

The table *Architecture versions* on page xvi lists the standard names of the current (not obsolete) ARM/Thumb architecture versions described in this manual. These names provide a shorthand way of describing the precise instruction set implemented by an ARM processor. However, this manual normally uses descriptive phrases such as "*M variants of architecture version 3 and above*" to avoid the use of lists of architecture names which are already long and will grow further in the future.

Obsolete architecture names are ARMv1, ARMv2, ARMv2a, and ARMv3G. These are the versions 1, 2, 2a, and 3G described in *Architecture versions and variants* on page xi.

Architecture versions

Name	ARM instruction set version	Thumb instruction set version	Long multiply instructions?	Enhanced DSP instructions
ARMv3	3	None	No	No
ARMv3M	3	None	Yes	No
ARMv4xM	4	None	No	No
ARMv4	4	None	Yes	No
ARMv4TxM	4	1	No	No
ARMv4T	4	1	Yes	No
ARMv5xM	5	None	No	No
ARMv5	5	None	Yes	No
ARMv5TxM	5	2	No	No
ARMv5T	5	2	Yes	No
ARMv5TExP	5	2	Yes	All but `LDRD`, `MCRR`, `MRRC`, `PLD`, and `STRD`
ARMv5TE	5	2	Yes	Yes

ARM DDI 0100F-02

Using this manual

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

Part A - CPU Architectures

Part A describes the ARM and Thumb instruction sets, and contains the following chapters:

Chapter A1   Gives a brief overview of the ARM instruction set.

Chapter A2   Describes the types of value that ARM instructions operate on, the general-purpose registers that contain those values, and the Program Status Registers. This chapter also describes how ARM processors handle interrupts and other exceptions, and contains general information about the memory interface of an ARM processor.

Chapter A3   Gives a description of the ARM instruction set, organized by type of instruction.

Chapter A4   Contains detailed reference material on each ARM instruction, arranged alphabetically by instruction mnemonic.

Chapter A5   Contains detailed reference material on the addressing modes used by ARM instructions. The term *addressing mode* is interpreted broadly in this manual, to mean a procedure shared by many different instructions, for generating values used by the instructions. For four of the addressing modes described in this chapter, the values generated are memory addresses (which is the traditional role of an addressing mode). The remaining addressing mode generates values to be used as operands by data-processing instructions.

Chapter A6   Gives a description of the Thumb instruction set, organized by type of instruction. This chapter also contains information about how to switch between the ARM and Thumb instruction sets, and how exceptions that arise during Thumb state execution are handled.

Chapter A7   Contains detailed reference material on each Thumb instruction, arranged alphabetically by instruction mnemonic.

Part B - Memory and System Architectures

Part B describes standard memory system features that are normally implemented by the System Control coprocessor (coprocessor 15) in an ARM-based system. It contains the following chapters:

Chapter B1   Gives a brief overview of this part of the manual.

Chapter B3   Gives a general description of the System Control coprocessor and its use.

Chapter B4   Describes the standard ARM memory and system architecture based on the use of a *Memory Management Unit* (MMU). (Chapter B3 and Chapter B6 are also relevant to this architecture.)

Chapter B5   Gives a description of the simpler standard ARM memory and system architecture based on the use of a Protection Unit. (Chapter B3 and Chapter B6 are also relevant to this architecture.)

Chapter B6   Gives a description of the standard ways to control caches and write buffers in ARM memory systems. This chapter is relevant both to systems based on an MMU and to systems based on a Protection Unit.

Part C - Vector Floating-point Architecture

Part C describes the *Vector Floating-point* (VFP) architecture. This is a coprocessor extension to the ARM architecture designed for high floating-point performance on typical graphics and DSP algorithms.

Chapter C1   Gives a brief overview of the VFP architecture and information about its compliance with the IEEE 754-1985 floating-point arithmetic standard.

Chapter C2   Describes the floating-point formats supported by the VFP instruction set, the floating-point general-purpose registers that hold those values, and the VFP system registers.

Chapter C3   Describes the VFP coprocessor instruction set, organized by type of instruction.

Chapter C4   Contains detailed reference material on the VFP coprocessor instruction set, organized alphabetically by instruction mnemonic.

Chapter C5   Contains detailed reference material on the addressing modes used by VFP instructions. One of these is a traditional addressing mode, generating addresses for load/store instructions. The remainder specify how the floating-point general-purpose registers and instructions can be used to hold and perform calculations on vectors of floating-point values.

Part D - Debug Architecture

Part D describes the debug architecture. This is a coprocessor extension to the ARM architecture designed for tbd.

Gives a brief overview of the debug architecture and information about tbd.

ARM DDI 0100F-02

Conventions

This manual employs typographic and other conventions intended to improve its ease of use.

General typographic conventions

typewriter	Is used for assembler syntax descriptions, pseudo-code descriptions of instructions, and source code examples. In the cases of assembler syntax descriptions and pseudo-code descriptions, see the additional conventions below.
	The typewriter font is also used in the main text for instruction mnemonics and for references to other items appearing in assembler syntax descriptions, pseudo-code descriptions of instructions and source code examples.
italic	Highlights important notes, introduces special terminology, and denotes internal cross-references and citations.
bold	Is used for emphasis in descriptive lists and elsewhere, where appropriate.
SMALL CAPITALS	Are used for a few terms which have specific technical meanings. Their meanings can be found in the *Glossary*.

Pseudo-code descriptions of instructions

A form of pseudo-code is used to provide precise descriptions of what instructions do. This pseudo-code is written in a typewriter font, and uses the following conventions for clarity and brevity:

- Indentation is used to indicate structure. For example, the range of statements that a for statement loops over, goes from the for statement to the next statement at the same or lower indentation level as the for statement (both ends exclusive).

- Comments are bracketed by /* and */, as in the C language.

- English text is occasionally used outside comments to describe functionality that is hard to describe otherwise.

- All keywords and special functions used in the pseudo-code are described in the *Glossary*.

- Assignment and equality tests are distinguished by using = for an assignment and == for an equality test, as in the C language.

- Instruction fields are referred to by the names shown in the encoding diagram for the instruction. When an instruction field denotes a register, a reference to it means the value in that register, rather than the register number, unless the context demands otherwise. For example, a Rn == 0 test is checking whether the value in the specified register is 0, but a Rd is R15 test is checking whether the specified register is register 15.

- When an instruction uses an addressing mode, the pseudo-code for that addressing mode generates one or more values that are used in the pseudo-code for the instruction. For example, the AND instruction described in *AND* on page A4-8 uses ARM addressing mode 1 (see *Addressing Mode 1 - Data-processing operands* on page A5-2). The pseudo-code for the addressing mode generates two values shifter_operand and shifter_carry_out, which are used by the pseudo-code for the AND instruction.

Assembler syntax descriptions

This manual contains numerous syntax descriptions for assembler instructions and for components of assembler instructions. These are shown in a `typewriter` font, and are as follows:

< > Any item bracketed by < and > is a short description of a type of value to be supplied by the user in that position. A longer description of the item is normally supplied by subsequent text. Such items often correspond to a similarly named field in an encoding diagram for an instruction. When the correspondence simply requires the binary encoding of an integer value or register number to be substituted into the instruction encoding, it is not described explicitly. For example, if the assembler syntax for an ARM instruction contains an item `<Rn>` and the instruction encoding diagram contains a 4-bit field named Rn, the number of the register specified in the assembler syntax is encoded in binary in the instruction field.

If the correspondence between the assembler syntax item and the instruction encoding is more complex than simple binary encoding of an integer or register number, the item description indicates how it is encoded.

{ } Any item bracketed by { and } is optional. A description of the item and of how its presence or absence is encoded in the instruction is normally supplied by subsequent text.

| This indicates an alternative character string. For example, `LDM|STM` is either `LDM` or `STM`.

spaces Single spaces are used for clarity, to separate items. When a space is obligatory in the assembler syntax, two or more consecutive spaces are used.

+/- This indicates an optional + or - sign. If neither is coded, + is assumed.

* When used in a combination like `<immed_8> * 4`, this describes an immediate value which must be a specified multiple of a value taken from a numeric range. In this instance, the numeric range is 0 to 255 (the set of values that can be represented as an 8-bit immediate) and the specified multiple is 4, so the value described must be a multiple of 4 in the range 4*0 = 0 to 4*255 = 1020.

All other characters must be encoded precisely as they appear in the assembler syntax. Apart from { and }, the special characters described above do not appear in the basic forms of assembler instructions documented in this manual. The { and } characters need to be encoded in a few places as part of a variable item. When this happens, the long description of the variable item indicates how they must be used.

——— **Note** ———

This manual only attempts to describe the most basic forms of assembler instruction syntax. In practice, assemblers normally recognize a much wider range of instruction syntaxes, as well as various directives to control the assembly process and additional features such as symbolic manipulation and macro expansion. All of these are beyond the scope of this manual.

For descriptions of the extra facilities provided by the assemblers included in ARM Development Systems, see the *ARM Software Development Toolkit Reference Guide* (ARM DUI 0041) for SDT 2.50, or the *ARM Developer Suite Tools Guide* (ARM DUI 0067) for ADS 1.0.

Further reading

This section lists publications from both ARM Limited and third parties that provide additional information on the ARM family of processors.

ARM periodically provides updates and corrections to its documentation. See `http://www.arm.com` for current errata sheets and addenda, and the ARM Frequently Asked Questions.

ARM publications

This book contains TBD

Other publications

The following books are referred to in this manual, or provide additional information:

*   *IEEE Standard for Shared-Data Formats Optimized for Scalable Coherent Interface (SCI) Processors*, IEEE Std 1596.5-1993, ISBN 1-55937-354-7, IEEE).

*   The Java™ Virtual Machine Specification Second Edition, Tim Lindholm and Frank Yellin, published by Addison Wesley (ISBN: 0-201-43294-3)

Preface

ARM DDI 0100F-02

Part A
CPU Architecture

Chapter A1
Introduction to the ARM Architecture

This chapter introduces the ARM architecture and contains the following sections:

- *About the ARM architecture* on page A1-2
- *ARM instruction set* on page A1-6
- *Thumb instruction set* on page A1-11.

A1.1 About the ARM architecture

The ARM architecture has evolved to a point where it supports implementations across a wide spectrum of performance points. Over a billion parts have shipped, establishing it as the dominant architecture across many market segments. The architectural simplicity of ARM processors has traditionally lead to very small implementations, and small implementations allow devices with very low power consumption. Implementation size, performance, and very low power consumption remain key attributes in the development of the ARM architecture.

The ARM is a *Reduced Instruction Set Computer* (RISC), as it incorporates these typical RISC architecture features:

• a large uniform register file

• a *load/store* architecture, where data-processing operations only operate on register contents, not directly on memory contents

• simple addressing modes, with all load/store addresses being determined from register contents and instruction fields only

• uniform and fixed-length instruction fields, to simplify instruction decode.

In addition, the ARM architecture provides:

• control over both the *Arithmetic Logic Unit* (ALU) and shifter in every data-processing instruction to maximize the use of an ALU and a shifter

• auto-increment and auto-decrement addressing modes to optimize program loops

• Load and Store Multiple instructions to maximize data throughput

• conditional execution of almost all instructions to maximize execution throughput.

These enhancements to a basic RISC architecture allow ARM processors to achieve a good balance of high performance, small code size, low power consumption, and small silicon area.

A1.1.1 ARM registers

ARM has 31 general-purpose 32-bit registers. At any one time, 16 of these registers are visible. The other registers are used to speed up exception processing. All the register specifiers in ARM instructions can address any of the 16 visible registers.

The main bank of 16 registers is used by all unprivileged code. These are the User mode registers. User mode is different from all other modes as it is unprivileged, which means:

• User mode is the only mode which cannot switch to another processor mode without generating an exception

• memory systems and coprocessors might allow User mode less access to memory and coprocessor functionality than a privileged mode.

Two of the 16 visible registers have special roles:

Link register   Register 14 is the *Link Register* (LR). This register holds the address of the next instruction after a Branch and Link (BL or BLX) instruction, which is the instruction used to make a subroutine call. At all other times, R14 can be used as a general-purpose register.

Program counter   Register 15 is the *Program Counter* (PC). It can be used in most instructions as a pointer to the instruction which is two instructions after the instruction being executed. In ARM state, all ARM instructions are four bytes long (one 32-bit word) and are always aligned on a word boundary. This means that the bottom two bits of the PC are always zero, and therefore the PC contains only 30 non-constant bits. Two other processor states are supported by some versions of the architecture. Thumb state is supported on T variants, and Java state on J variants. The PC can be halfword (16-bit) and byte aligned respectively in these states.

The remaining 14 registers have no special hardware purpose. Their uses are defined purely by software. Software normally uses R13 as a *Stack Pointer* (SP).

For more details on registers, refer to *Registers* on page A2-4.

A1.1.2   Exceptions

ARM supports five types of exception, and a privileged processing mode for each type. The five types of exceptions are:

*   fast interrupt
*   normal interrupt
*   memory aborts, which can be used to implement memory protection or virtual memory
*   attempted execution of an undefined instruction
*   software interrupt (SWI) instructions which can be used to make a call to an operating system.

When an exception occurs, some of the standard registers are replaced with registers specific to the exception mode. All exception modes have replacement *banked* registers for R13 and R14. The fast interrupt mode has more registers for fast interrupt processing.

When an exception handler is entered, R14 holds the return address for exception processing. This is used to return after the exception is processed and to address the instruction that caused the exception.

Register 13 is banked across exception modes to provide each exception handler with a private stack pointer. The fast interrupt mode also banks registers 8 to 12 so that interrupt processing can begin without the need to save or restore these registers.

There is a sixth privileged processing mode, System mode, which uses the User mode registers. This is used to run tasks that require privileged access to memory and/or coprocessors, without limitations on which exceptions can occur during the task.

In addition to the above, reset shares the same privileged mode as SWIs.

For more details on exceptions, refer to *Exceptions* on page A2-15.

The exception process

When an exception occurs, the ARM processor halts execution after the current instruction and begins execution at one of a number of fixed addresses in memory, known as the *exception vectors*. There is a separate vector location for each exception, including reset.

An operating system installs a handler on every exception at initialization. Privileged operating system tasks are normally run in System mode to allow exceptions to occur within the operating system without state loss.

A1.1.3 Status registers

All processor state other than the general-purpose register contents is held in *status registers*. The current operating processor status is in the *Current Program Status Register* (CPSR). The CPSR holds:

- 4 condition code flags (Negative, Zero, Carry and oVerflow).
- 1 sticky (Q) flag (ARM architecture v5 and above only). This encodes whether saturation has occurred in saturated arithmetic instructions, or signed overflow in some specific multiply accumulate instructions.
- 4 GE (Greater than or Equal) flags (ARM architecture v6 and above only). These encode the following conditions separately for each operation in parallel instructions:
    — whether the results of signed operations were non-negative
    — whether unsigned operations produced a carry or a borrow.
- 3 interrupt disable bits, one for each type of interrupt (2 in ARM architecture v5 and below).
- 5 bits that encode the current processor mode.
- 2 bits that encode whether ARM, Thumb, or Java instructions are being executed.
- 1 bit that controls the endianness of load and store operations (ARM architecture v6 and above only).

Each exception mode also has a *Saved Program Status Register* (SPSR) which holds the CPSR of the task immediately before the exception occurred. The CPSR and the SPSRs are accessed with special instructions.

For more details on status registers, refer to *Program status registers* on page A2-10.

Table A1-1 Status register summary

Field	Description	Architecture
N Z C V	Condition code flags	All
J	Java state flag	5TEJ and above
GE[3:0]	SIMD condition flags	6 and above
E	Endian Load/Store	6 and above
A	Imprecise Abort Mask	6 and above
I	IRQ Interrupt Mask	All
F	FIQ Interrupt Mask	All
T	Thumb state flag	4T and above
Mode[4:0]	Processor mode	All

A1.2 ARM instruction set

The ARM instruction set can be divided into six broad classes of instruction:

*   *Branch instructions*
*   *Data-processing instructions* on page A1-7
*   *Status register transfer instructions* on page A1-8
*   *Load and store instructions* on page A1-8
*   *Coprocessor instructions* on page A1-10
*   *Exception-generating instructions* on page A1-10.

Most data-processing instructions and one type of coprocessor instruction can update the four condition code flags in the CPSR (Negative, Zero, Carry and oVerflow) according to their result.

Almost all ARM instructions contain a 4-bit *condition* field. One value of this field specifies that the instruction is executed unconditionally.

Fourteen other values specify *conditional execution* of the instruction. If the condition code flags indicate that the corresponding condition is true when the instruction starts executing, it executes normally. Otherwise, the instruction does nothing. The 14 available conditions allow:

*   tests for equality and non-equality
*   tests for <, <=, >, and >= inequalities, in both signed and unsigned arithmetic
*   each condition code flag to be tested individually.

The sixteenth value of the condition field encodes alternative instructions. These do not allow conditional execution. Before ARMv5 these instructions were UNPREDICTABLE.

A1.2.1 Branch instructions

As well as allowing many data-processing or load instructions to change control flow by writing the PC, a standard Branch instruction is provided with a 24-bit signed word offset, allowing forward and backward branches of up to 32MB.

There is a Branch and Link (BL) option that also preserves the address of the instruction after the branch in R14, the LR. This provides a subroutine call which can be returned from by copying the LR into the PC.

There are also branch instructions which can switch instruction set, so that execution continues at the branch target using the Thumb or Java instruction sets. Thumb support allows ARM code to call Thumb subroutines, and ARM subroutines to return to a Thumb caller. Similar instructions in the Thumb instruction set allow the corresponding Thumb → ARM switches. An overview of the Thumb instruction set is provided in Chapter A6 *The Thumb Instruction Set*. The BXJ instruction introduced with the J variant of ARMv5, and present in ARMv6, provides the architected mechanism for entry to Java state, and the associated assertion of the J flag in the CPSR.

A1.2.2 Data-processing instructions

The data-processing instructions perform calculations on the general-purpose registers. There are five types of data-processing instructions:

* *Arithmetic/logic instructions*
* *Comparison instructions*
* *Single Instruction Multiple Data (SIMD) Instructions*
* *Multiply instructions* on page A1-8
* *Miscellaneous Data Processing instructions* on page A1-8.

Arithmetic/logic instructions

The following arithmetic/logic instructions share a common instruction format. These perform an arithmetic or logical operation on up to two source operands, and write the result to a destination register. They can also optionally update the condition code flags, based on the result.

Of the two source operands:

* one is always a register
* the other has two basic forms:
    — an immediate value
    — a register value, optionally shifted.

If the operand is a shifted register, the shift amount can be either an immediate value or the value of another register. Four types of shift can be specified. Every arithmetic/logic instruction can therefore perform an arithmetic/logic and a shift operation. As a result, ARM does not have dedicated shift instructions.

Because the *Program Counter* (PC) is a general-purpose register, arithmetic/logic instructions can write their results directly to the PC. This allows easy implementation of a variety of jump instructions.

Comparison instructions

The comparison instructions use the same instruction format as the arithmetic/logic instructions. These perform an arithmetic or logical operation on two source operands, but do not write the result to a register. They always update the condition flags, based on the result.

The source operands of comparison instructions take the same forms as those of arithmetic/logic instructions, including the ability to incorporate a shift operation.

Single Instruction Multiple Data (SIMD) Instructions

The add and subtract instructions treat each operand as two parallel 16-bit numbers, or four parallel 8-bit numbers. They can be treated as signed or unsigned. The operations can optionally be saturating, wrap around, or the results can be halved to avoid overflow.

These instructions are available in ARM architecture v6 and above.

Multiply instructions

There are several classes of multiply instructions, introduced at different times into the architecture. See *Multiply instructions* on page A3-14 for details.

Miscellaneous Data Processing instructions

These include Count Leading Zeros (`CLZ`) and Unsigned Sum of Absolute Differences with optional Accumulate (`USAD8` and `USADA8`).

A1.2.3   Status register transfer instructions

The status register transfer instructions transfer the contents of the CPSR or an SPSR to or from a general-purpose register. Writing to the CPSR can:

*   set the values of the condition code flags
*   set the values of the interrupt enable bits
*   set the processor mode
*   alter the endianness of Load and Store operations.

A1.2.4   Load and store instructions

The following load and store instructions are available:

*   *Load and Store Register*
*   *Load and Store Multiple registers* on page A1-9
*   *Load and Store Register Exclusive* on page A1-9
*   *Swap register and memory contents* on page A1-9.

Load and Store Register

Load Register instructions can load a 64-bit doubleword, a 32-bit word, a 16-bit halfword, or an 8-bit byte from memory into a register or registers. Byte and halfword loads can be automatically zero-extended or sign-extended as they are loaded.

Store Register instructions can store a 64-bit doubleword, a 32-bit word, a 16-bit halfword, or an 8-bit byte from a register or registers to memory.

From ARMv6, unaligned loads and stores of words and halfwords are supported.

Load and Store Register instructions have three primary addressing modes, all of which use a *base register* and an *offset* specified by the instruction:

*   In *offset addressing*, the memory address is formed by adding or subtracting an offset to or from the base register value.

*   In *pre-indexed addressing*, the memory address is formed in the same way as for offset addressing. As a side effect, the memory address is also written back to the base register.

- In *post-indexed addressing*, the memory address is the base register value. As a side effect, an offset is added to or subtracted from the base register value and the result is written back to the base register.

In each case, the offset can be either an immediate or the value of an *index register*. Register-based offsets can also be scaled with shift operations.

As the PC is a general-purpose register, a 32-bit value can be loaded directly into the PC to perform a jump to any address in the 4GB memory space.

Load and Store Multiple registers

Load Multiple (LDM) and Store Multiple (STM) instructions perform a block transfer of any number of the general-purpose registers to or from memory. Four addressing modes are provided:

- pre-increment
- post-increment
- pre-decrement
- post-decrement.

The base address is specified by a register value, which can be optionally updated after the transfer. As the subroutine return address and PC values are in general-purpose registers, very efficient subroutine entry and exit sequences can be constructed with LDM and STM:

- A single STM instruction at subroutine entry can push register contents and the return address onto the stack, updating the stack pointer in the process.

- A single LDM instruction at subroutine exit can restore register contents from the stack, load the PC with the return address, and update the stack pointer.

LDM and STM instructions also allow very efficient code for block copies and similar data movement algorithms.

Load and Store Register Exclusive

These instructions support cooperative memory synchronization. They are designed to provide the atomic behavior required for semaphores without locking all system resources between the load and store phases. See *LDREX* on page A4-51 and *STREX* on page A4-208 for details.

Swap register and memory contents

A swap (SWP) instruction performs the following sequence of operations:

1. It loads a value from a register-specified memory location.
2. It stores the contents of a register to the same memory location.
3. It writes the value loaded in step 1 to a register.

By specifying the same register for steps 2 and 3, the contents of a memory location and a register are interchanged.

The swap operation performs a special indivisible bus operation that allows atomic update of semaphores. Both 32-bit word and 8-bit byte semaphores are supported.

A1.2.5 Coprocessor instructions

There are three types of coprocessor instructions:

Data-processing instructions

These start a coprocessor-specific internal operation.

Data transfer instructions

These transfer coprocessor data to or from memory. The address of the transfer is calculated by the ARM processor.

Register transfer instructions

These allow a coprocessor value to be transferred to or from an ARM register, or a pair of ARM registers.

A1.2.6 Exception-generating instructions

Two types of instruction are designed to cause specific exceptions to occur.

Software interrupt instructions

SWI instructions cause a software interrupt exception to occur. These are normally used to make calls to an operating system, to request an OS-defined service. The exception entry caused by a SWI instruction also changes to a privileged processor mode. This allows an unprivileged task to gain access to privileged functions, but only in ways permitted by the OS.

Software breakpoint instructions

BKPT instructions cause an abort exception to occur. If suitable debugger software is installed on the abort vector, an abort exception generated in this fashion is treated as a breakpoint. If debug hardware is present in the system, it can instead treat a BKPT instruction directly as a breakpoint, preventing the abort exception from occurring.

In addition to the above, the following types of instruction cause an Undefined Instruction exception to occur:
* coprocessor instructions which are not recognized by any hardware coprocessor
* most instruction words that have not yet been allocated a meaning as an ARM instruction.

In each case, this exception is normally used either to generate a suitable error or to initiate software emulation of the instruction.

A1.3   Thumb instruction set

The Thumb instruction set is a subset of the ARM instruction set, with each instruction encoded in 16 bits instead of 32 bits. For details see Chapter A6 *The Thumb Instruction Set*.

Chapter A2
Programmers' Model

This chapter introduces the ARM programmer's model. It contains the following sections:

- *Data types* on page A2-2
- *Processor modes* on page A2-3
- *Registers* on page A2-4
- *General-purpose registers* on page A2-6
- *Program status registers* on page A2-10
- *Exceptions* on page A2-15
- *Endian Support* on page A2-29
- *Unaligned access support* on page A2-42.

A2.1 Data types

ARM processors support the following data types:

Byte   8 bits.

Halfword   16 bits.

Word   32 bits.

Doubleword   64 bits, treated as two 32-bit words by load and store operations.

——— **Note** ———

- Support for halfwords was introduced in version 4. Support for doublewords was introduced in ARMv5E. ARMv6 has introduced unaligned data support for words and halfwords. See *Unaligned access support* on page A2-42 for more information.

- When any of these types is described as *unsigned*, the N-bit data value represents a non-negative integer in the range 0 to $+2^N-1$, using normal binary format.

- When any of these types is described as *signed*, the N-bit data value represents an integer in the range -2^{N-1} to $+2^{N-1}-1$, using two's complement format.

- Most data operations, for example ADD, are performed on word quantities. Long multiplies support 64-bit results with or without accumulation. ARMv5E introduced some halfword multiply operations. ARMv6 introduced a variety of Single Instruction Multiple Data (SIMD) instructions operating on two halfwords or four bytes in parallel.

- Load and store operations can transfer bytes, halfwords, words, or doublewords to and from memory, automatically zero-extending or sign-extending bytes or halfwords as they are loaded.

- ARM instructions are exactly one word (and are aligned on a four-byte boundary). Thumb instructions are exactly one halfword (and are aligned on a two-byte boundary). Java instructions are byte aligned.

————————

A2.2   Processor modes

The ARM architecture supports the seven processor modes shown in Table A2-1.

Table A2-1 ARM processor modes

Processor mode			Description
User	usr	0b10000	Normal program execution mode
FIQ	fiq	0b10001	Supports a high-speed data transfer or channel process
IRQ	irq	0b10010	Used for general-purpose interrupt handling
Supervisor	svc	0b10011	A protected mode for the operating system
Abort	abt	0b10111	Implements virtual memory and/or memory protection
Undefined	und	0b11011	Supports software emulation of hardware coprocessors
System	sys	0b11111	Runs privileged operating system tasks (ARMv4 and above)

Mode changes can be made under software control, or can be caused by external interrupts or exception processing.

Most application programs execute in User mode. When the processor is in User mode, the program being executed is unable to access some protected system resources or to change mode, other than by causing an exception to occur (see *Exceptions* on page A2-15). This allows a suitably written operating system to control the use of system resources.

The modes other than User mode are known as *privileged modes*. They have full access to system resources and can change mode freely. Five of them are known as *exception modes*:

*   FIQ
*   IRQ
*   Supervisor
*   Abort
*   Undefined.

These are entered when specific exceptions occur. Each of them has some additional registers to avoid corrupting User mode state when the exception occurs (see *Registers* on page A2-4 for details).

The remaining mode is System mode, and is only present in ARM architecture version 4 and above. It is not entered by any exception and has exactly the same registers available as User mode. However, it is a privileged mode and is therefore not subject to the User mode restrictions. It is intended for use by operating system tasks that need access to system resources, but wish to avoid using the additional registers associated with the exception modes. Avoiding such use ensures that the task state is not corrupted by the occurrence of any exception.

A2.3   Registers

The ARM processor has a total of 37 registers:

*   31 general-purpose registers, including a program counter. These registers are 32 bits wide and are described in *General-purpose registers* on page A2-6.
*   6 status registers. These registers are also 32 bits wide, but only a subset of the 32 bits are allocated or need to be implemented. The subset depends on the architecture variant supported. These are described in *Program status registers* on page A2-10.

Registers are arranged in partially overlapping banks, with a different register bank for each processor mode, as shown in Figure A2-1 on page A2-5. At any time, 15 general-purpose registers (R0 to R14), one or two status registers and the program counter are visible. Each column of Figure A2-1 on page A2-5 shows which general-purpose and status registers are visible in the indicated processor mode.

Modes						
	Privileged modes					
		Exception modes				
User	**System**	**Supervisor**	**Abort**	**Undefined**	**Interrupt**	**Fast interrupt**
R0	R0	R0	R0	R0	R0	R0
R1	R1	R1	R1	R1	R1	R1
R2	R2	R2	R2	R2	R2	R2
R3	R3	R3	R3	R3	R3	R3
R4	R4	R4	R4	R4	R4	R4
R5	R5	R5	R5	R5	R5	R5
R6	R6	R6	R6	R6	R6	R6
R7	R7	R7	R7	R7	R7	R7
R8	R8	R8	R8	R8	R8	R8_fiq
R9	R9	R9	R9	R9	R9	R9_fiq
R10	R10	R10	R10	R10	R10	R10_fiq
R11	R11	R11	R11	R11	R11	R11_fiq
R12	R12	R12	R12	R12	R12	R12_fiq
R13	R13	R13_svc	R13_abt	R13_und	R13_irq	R13_fiq
R14	R14	R14_svc	R14_abt	R14_und	R14_irq	R14_fiq
PC	PC	PC	PC	PC	PC	PC

CPSR	CPSR	CPSR	CPSR	CPSR	CPSR	CPSR
		SPSR_svc	SPSR_abt	SPSR_und	SPSR_irq	SPSR_fiq

indicates that the normal register used by User or System mode has been replaced by an alternative register specific to the exception mode

Figure A2-1 Register organization

A2.4   General-purpose registers

The general-purpose registers R0-R15 can be split into three groups. These groups differ in the way they are banked and in their special-purpose uses:

*   *The unbanked registers, R0-R7*
*   *The banked registers, R8-R14*
*   Register 15, the PC, is described in *The program counter, R15* on page A2-8.

A2.4.1   The unbanked registers, R0-R7

Registers R0 to R7 are *unbanked registers*. This means that each of them refers to the same 32-bit physical register in all processor modes. They are completely general-purpose registers, with no special uses implied by the architecture, and can be used wherever an instruction allows a general-purpose register to be specified.

A2.4.2   The banked registers, R8-R14

Registers R8 to R14 are *banked registers*. The physical register referred to by each of them depends on the current processor mode. Where a particular physical register is intended, without depending on the current processor mode, a more specific name (as described below) is used. Almost all instructions allow the banked registers to be used wherever a general-purpose register is allowed.

——— **Note** ———

A few exceptions to this rule are noted in the individual instruction descriptions. Where a restriction exists on the use of banked registers, it always applies to all of R8 to R14. For example, R8 to R12 are subject to such restrictions even in systems in which FIQ mode is never used and so only one physical version of the register is ever in use.

Registers R8 to R12 have two banked physical registers each. One is used in all processor modes other than FIQ mode, and the other is used in FIQ mode. Where it is necessary to be specific about which version is being referred to, the first group of physical registers are referred to as R8_usr to R12_usr and the second group as R8_fiq to R12_fiq.

Registers R8 to R12 do not have any dedicated special purposes in the architecture. However, for interrupts that are simple enough to be processed using registers R8 to R14 only, the existence of separate FIQ mode versions of these registers allows very fast interrupt processing.

Registers R13 and R14 have six banked physical registers each. One is used in User and System modes, and each of the remaining five is used in one of the five exception modes. Where it is necessary to be specific about which version is being referred to, you use names of the form:

```
R13_<mode>
R14_<mode>
```

where `<mode>` is the appropriate one of `usr`, `svc` (for Supervisor mode), `abt`, `und`, `irq` and `fiq`.

Register R13 is normally used as a stack pointer and is also known as the SP. The SRS instruction, introduced in ARMv6, is the only ARM instruction that uses R13 in a special-case manner. There are such instructions in the Thumb instruction set, as described in Chapter A6 *The Thumb Instruction Set*.

Each exception mode has its own banked version of R13, that should normally be initialized to point to a stack dedicated to that exception mode. On entry, the exception handler typically stores to this stack the values of other registers to be used. By reloading these values into the registers when it returns, the exception handler can ensure that it does not corrupt the state of the program that was being executed when the exception occurred.

Register R14 (also known as the *Link Register* or LR) has two special functions in the architecture:

• In each mode, the mode's own version of R14 is used to hold subroutine return addresses. When a subroutine call is performed by a BL or BLX instruction, R14 is set to the subroutine return address. The subroutine return is performed by copying R14 back to the program counter. This is typically done in one of the two following ways:

— Execute either of these instructions:

```
MOV PC,LR
BX LR
```

— On subroutine entry, store R14 to the stack with an instruction of the form:

```
STMFD SP!,{<registers>,LR}
```

and use a matching instruction to return:

```
LDMFD SP!,{<registers>,PC}
```

• When an exception occurs, the appropriate exception mode's version of R14 is set to the exception return address (offset by a small constant for some exceptions). The exception return is performed in a similar way to a subroutine return, but using slightly different instructions to ensure full restoration of the state of the program that was being executed when the exception occurred. See *Exceptions* on page A2-15 for more details.

Register R14 can be treated as a general-purpose register at all other times.

——— **Note** ———

When nested exceptions are possible, the two special-purpose uses might conflict. For example, if an IRQ interrupt occurs when a program is being executed in User mode, none of the User mode registers are necessarily corrupted. But if an interrupt handler running in IRQ mode re-enables IRQ interrupts and a nested IRQ interrupt occurs, any value the outer interrupt handler is holding in R14_irq at the time is overwritten by the return address of the nested interrupt.

System programmers need to be careful about such interactions. The usual way to deal with them is to ensure that the appropriate version of R14 does not hold anything significant at times that nested exceptions can occur. When this is hard to do in a straightforward way, it is usually best to change to another processor mode during entry to the exception handler, before re-enabling interrupts or otherwise allowing nested exceptions to occur. (In ARM architecture version 4 and above, System mode is usually the best mode to use for this purpose.)

A2.4.3 The program counter, R15

Register R15 holds the *Program Counter* (PC). It can often be used in place of one of the general-purpose registers R0 to R14, and is therefore considered one of the general-purpose registers. However, there are also many instruction-specific restrictions or special cases about its use. These are noted in the individual instruction descriptions. Usually, the instruction is UNPREDICTABLE if R15 is used in a manner that breaks these restrictions.

The Program Counter is always used for a special purpose, as described in:

* *Reading the program counter*
* *Writing the program counter* on page A2-9.

Reading the program counter

When an instruction reads R15 without breaking any of the restrictions on its use, the value read is the address of the instruction plus 8 bytes. As ARM instructions are always word-aligned, bits[1:0] of the resulting value are always zero. (In T variants of the architecture, this behavior changes during Thumb state execution - see Chapter A6 *The Thumb Instruction Set* for details.)

This way of reading the PC is primarily used for quick, position-independent addressing of nearby instructions and data, including position-independent branching within a program.

An exception to the above rule occurs when an STR or STM instruction stores R15. Such instructions can store either the address of the instruction plus 8 bytes, like other instructions that read R15, or the instruction's own address plus 12 bytes. Whether the offset of 8 or the offset of 12 is used is IMPLEMENTATION DEFINED. An implementation must use the same offset for all STR and STM instructions that store R15. It cannot use 8 for some of them and 12 for others.

Because of this exception, it is usually best to avoid the use of STR and STM instructions that store R15. If this is difficult, use a suitable instruction sequence in the program to ascertain which offset the implementation uses. For example, if R0 points to an available word of memory, then the following instructions put the offset of the implementation in R0:

```
SUB  R1, PC, #4    ; R1 = address of following STR instruction
STR  PC, [R0]      ; Store address of STR instruction + offset,
LDR  R0, [R0]      ; then reload it
SUB  R0, R0, R1    ; Calculate the offset as the difference
```

——— **Note** ———

The rules about how R15 is read apply only to reads by instructions. In particular, they do not necessarily describe the values placed on a hardware address bus during instruction fetches. Like all other details of hardware interfaces, such values are IMPLEMENTATION DEFINED.

Writing the program counter

When an instruction writes R15 without breaking any of the restrictions on its use, the normal result is that the value written to R15 is treated as an instruction address and a branch occurs to that address.

Since ARM instructions are required to be word-aligned, values written to R15 are normally expected to have bits[1:0] == 0b00. The precise rules for this depend on the architecture version:

*   In ARM architecture versions 4 and above, bits[1:0] of a value written to R15 in ARM state must be 0b00. If they are not, the results are UNPREDICTABLE.

Similarly, in T variants of ARM architecture versions 4 and above, Thumb instructions are required to be halfword-aligned. Bit[0] of a value written to R15 in Thumb state is ignored, so that the actual destination address of the instruction is (value written to R15) AND 0xFFFFFFFE.

Several instructions have their own rules for interpreting values written to R15. For example, BX and other instructions designed to transfer between ARM and Thumb states use bit[0] of the value to select whether to execute the code at the destination address in ARM state or Thumb state. Special rules of this type are described on the individual instruction pages, and override the general rules in this section.

The program counter in ARMv6

The rules relating to low-order bits of the PC are revised from ARMv6:

*   When R15 is read in ARM state, bits[1:0] must read as 0b00.

*   When R15 is read in Thumb state, bit[0] must read as 0.

*   When R15 is written in ARM state, bits[1:0] of the value written must be ignored. Bit[1] of the PC must be written as zero. If bit[0] of the PC is implemented (which depends on how the Java Extension is implemented), then zero must be written to it.

*   When R15 is written in Thumb state, bit[0] of the value written must be ignored. If bit[0] of the PC is implemented (which depends on how the Java Extension is implemented), then zero must be written to it.

These rules relate to register reads and writes of the PC, and do not necessarily restrict hardware interfaces, which are IMPLEMENTATION DEFINED. Unaligned address values may be sent to the memory interface where the system is responsible for alignment correctness of the associated instruction fetch.

In addition, rules in existing instruction descriptions (such as BX, BLX, LDR or LDM (1) supporting ARM to Thumb state transition) relating to how they write values to the PC, continue to override the general rules where necessary. See the individual instruction descriptions for the details.

A2.5   Program status registers

The *current program status register* (CPSR) is accessible in all processor modes. It contains condition code flags, interrupt disable bits, the current processor mode, and other status and control information. Each exception mode also has a *saved program status register* (SPSR), that is used to preserve the value of the CPSR when the associated exception occurs.

───── **Note** ─────

User mode and System mode do not have an SPSR, because they are not exception modes. All instructions that read or write the SPSR are UNPREDICTABLE when executed in User mode or System mode.

────────────

The format of the CPSR and the SPSRs is shown below.

31	30	29	28	27	26 25	24	23 20	19 16	15 10	9	8	7	6	5	4 0
N	Z	C	V	Q	Res	J	Reserved	GE[3:0]	Reserved	E	A	I	F	T	M[4:0]

A2.5.1   Types of PSR bits

There are four categories of PSR bits, according to rules about updating them:

Unallocated bits   These are reserved for future expansion. Implementations must read these bits as 0 and ignore writes to them. For maximum compatibility with future extensions to the architecture, you must write them with values read from the same bits.

User-writable bits   These can be written from any mode. The N, Z, C, V, Q, GE[3:0], and E bits are user-writable.

Privileged bits   These can be written from any privileged mode. Writes to privileged bits in User mode are ignored. The A, I, F, and M[4:0] bits are privileged.

Execution state bits These cannot be written from User mode. Any attempt to write them from User mode is ignored. They can be written from any privileged mode. They are always 0 in ARM state. The J and T bits are execution state bits.

A2.5.2   The condition code flags

The N, Z, C, and V (Negative, Zero, Carry and oVerflow) bits are collectively known as the *condition code flags*, often referred to as *flags*. The condition code flags in the CPSR can be tested by most instructions to determine whether the instruction is to be executed.

The condition code flags are usually modified by:

•   Execution of a comparison instruction (CMN, CMP, TEQ or TST).

- Execution of some other arithmetic, logical or move instruction, where the destination register of the instruction is not R15. Most of these instructions have both a flag-preserving and a flag-setting variant, with the latter being selected by adding an S qualifier to the instruction mnemonic. Some of these instructions only have a flag-preserving version. This is noted in the individual instruction descriptions.

In either case, the new condition code flags (after the instruction has been executed) usually mean:

N   Is set to bit 31 of the result of the instruction. If this result is regarded as a two's complement signed integer, then N = 1 if the result is negative and N = 0 if it is positive or zero.

Z   Is set to 1 if the result of the instruction is zero (this often indicates an *equal* result from a comparison), and to 0 otherwise.

C   Is set in one of four ways:

- For an addition, including the comparison instruction CMN, C is set to 1 if the addition produced a carry (that is, an unsigned overflow), and to 0 otherwise.

- For a subtraction, including the comparison instruction CMP, C is set to 0 if the subtraction produced a borrow (that is, an unsigned underflow), and to 1 otherwise.

- For non-addition/subtractions that incorporate a shift operation, C is set to the last bit shifted out of the value by the shifter.

- For other non-addition/subtractions, C is normally left unchanged (but see the individual instruction descriptions for any special cases).

V   Is set in one of two ways:

- For an addition or subtraction, V is set to 1 if signed overflow occurred, regarding the operands and result as two's complement signed integers.

- For non-addition/subtractions, V is normally left unchanged (but see the individual instruction descriptions for any special cases).

The flags can be modified in these additional ways:

- Execution of an MSR instruction, as part of its function of writing a new value to the CPSR or SPSR.

- Execution of MRC instructions with destination register R15. The purpose of such instructions is to transfer coprocessor-generated condition code flag values to the ARM processor.

- Execution of some variants of the LDM instruction. These variants copy the SPSR to the CPSR, and their main intended use is for returning from exceptions.

- Execution of flag-setting variants of arithmetic and logical instructions whose destination register is R15. These also copy the SPSR to the CPSR, and are mainly intended for returning from exceptions.

The Q flag

In E variants of ARM architecture 5 and above, bit[27] of the CPSR is known as the Q flag and is used to indicate whether overflow and/or saturation has occurred in some of the tbd terminology? enhanced DSP instructions. Similarly, bit[27] of each SPSR is a Q flag, and is used to preserve and restore the CPSR Q flag if an exception occurs.

In architecture versions prior to version 5, and in non-E variants of architecture version 5, bit[27] of the CPSR and SPSRs should be treated as described in *Other bits* on page A2-14.

The GE[3:0] bits

In ARM architecture v6, the SIMD instructions use bits[19:16] as *Greater than or Equal* (GE) flags for individual bytes or halfwords of the result. You can use these flags to control a later SEL instruction, see *SEL* on page A4-134 for further details.

Instructions that operate on halfwords:
*   set or clear GE[3:2] together, based on the result of the top halfword calculation
*   set or clear GE[1:0] together, based on the result of the bottom halfword calculation.

Instructions that operate on bytes:
*   set or clear GE[3] according to the result of the top byte calculation
*   set or clear GE[2] according to the result of the second byte calculation
*   set or clear GE[1] according to the result of the third byte calculation
*   set or clear GE[0] according to the result of the bottom byte calculation.

Each bit is set if the results of the corresponding calculation are as follows:
*   for unsigned byte addition, if the result is greater than or equal to 2^8
*   for unsigned halfword addition, if the result is greater than or equal to 2^{16}
*   for unsigned subtraction, if the result is greater than or equal to zero
*   for signed arithmetic, if the result is greater than or equal to zero.

Each bit is clear otherwise.

In architecture versions prior to version 6, bits[19:16] of the CPSR and SPSRs should be treated as described in *Other bits* on page A2-14.

A2.5.3   The J bit

On J variants of ARM architecture 5 and above, the J bit has the following meaning:

J = 0   indicates ARM (T=0) or Thumb (T=1) execution

J = 1   indicates Java execution (the T bit must be zero)

The BXJ instruction is used to switch from ARM state into Java state where hardware acceleration is present and enabled. On processors without hardware acceleration or when it is disabled, the BXJ instruction behaves as a BX instruction, and the J-bit remains clear. See *BXJ* on page A4-20.

In implementations without the Java extension, bit[24] of the CPSR and SPSRs should be treated as described in *Other bits* on page A2-14.

A2.5.4   The E bit

From ARM architecture version 6, Bit[9] controls load and store endianness for data handling. See *Instructions to change CPSR E-bit* on page A2-41. This bit is ignored by instruction fetches.

In architecture versions prior to version 6, bit[9] of the CPSR and SPSRs must be treated as described in *Other bits* on page A2-14.

A2.5.5   The control bits

The bottom ten bits of a *Program Status Register* (PSR), incorporating A, I, F, T, and M[4:0], are known collectively as the *control bits*. The control bits change when an exception arises and can be altered by software only when the processor is in a privileged mode.

Interrupt disable bits

A, I, and F are the interrupt disable bits:

A bit   Disables imprecise Data Aborts when it is set. This is available only in architecture versions 6 and above. In earlier versions, bit[8] of CPSR and SPSRs must be treated as described in *Other bits* on page A2-14.

I bit   Disables IRQ interrupts when it is set.

F bit   Disables FIQ interrupts when it is set.

The T bit

The T bit *should be zero* (SBZ) on non-T variants of ARM architecture version 4. No instructions exist in these architectures that can switch between ARM and Thumb states.

On T variants of ARM architecture 4 and above, the T bit has the following meanings:

T = 0   Indicates ARM execution.

T = 1   Indicates Thumb execution.

Instructions that switch between ARM and Thumb states can be used freely on implementations of these architectures. All variants of the ARM architecture from version 6 shall support Thumb state operation.

On non-T variants of the ARM architecture, the T bit has the following meanings:

T = 0   Indicates ARM execution.

T = 1   Forces the next instruction executed to cause an undefined instruction exception (see *Undefined Instruction exception* on page A2-18).

Instructions that switch between ARM and Thumb states can be used on implementations of these architectures, but only function correctly as long as the program remains in ARM state. If the program attempts to switch to Thumb state, the first instruction executed after the attempted switch causes an undefined instruction exception. Entry into that exception then switches back to ARM state. The exception handler can detect that this was the cause of the exception from the fact that the T bit of SPSR_und is set.

Mode bits

M0, M1, M2, M3, and M4 (M[4:0]) are the mode bits. These determine the mode in which the processor operates. Their interpretation is shown in Table A2-2.

Table A2-2 The mode bits

M[4:0]	Mode	Accessible registers
0b10000	User	PC, R14 to R0, CPSR
0b10001	FIQ	PC, R14_fiq to R8_fiq, R7 to R0, CPSR, SPSR_fiq
0b10010	IRQ	PC, R14_irq, R13_irq, R12 to R0, CPSR, SPSR_irq
0b10011	Supervisor	PC, R14_svc, R13_svc, R12 to R0, CPSR, SPSR_svc
0b10111	Abort	PC, R14_abt, R13_abt, R12 to R0, CPSR, SPSR_abt
0b11011	Undefined	PC, R14_und, R13_und, R12 to R0, CPSR, SPSR_und
0b11111	System	PC, R14 to R0, CPSR (ARMv4 and above)

Not all combinations of the mode bits define a valid processor mode. Only those combinations explicitly described can be used. If any other value is programmed into the mode bits M[4:0], the result is UNPREDICTABLE.

A2.5.6 Other bits

Other bits in the program status registers are reserved for future expansion. In general, programmers must take care to write code in such a way that these bits are never modified. Failure to do this might result in code that has unexpected side effects on future versions of the architecture. See *Types of PSR bits* on page A2-10, and the usage notes for the MSR instruction on page A4-75 for more details.

A2.6   Exceptions

Exceptions are generated by internal and external sources to cause the processor to handle an event, such as an externally generated interrupt or an attempt to execute an undefined instruction. The processor state just before handling the exception must be preserved so that the original program can be resumed when the exception routine has completed. More than one exception can arise at the same time.

ARM supports seven types of exception. Table A2-3 lists the types of exception and the processor mode that is used to process that exception. When an exception occurs, execution is forced from a fixed memory address corresponding to the type of exception. These fixed addresses are called the *exception vectors*. ARMv6 introduces several extensions to the exception model:

- An imprecise abort mechanism which overloads the data memory abort vector and associated registers. This mechanism includes a mask bit (the A-bit) in CPSR/SPSR to ensure no loss of state information when a (precise) data memory abort occurs. The mechanism is described in *Imprecise aborts* on page A2-22.

- Support for vectored interrupts controlled by the VE bit in the system control coprocessor (see *Vectored interrupt support* on page A2-26). It is IMPLEMENTATION DEFINED whether support for this mechanism is included in earlier versions of the architecture.

- Support for a low interrupt latency configuration controlled by the FI bit in the system control coprocessor (see *Low interrupt latency configuration* on page A2-26). It is IMPLEMENTATION DEFINED whether support for this mechanism is included in earlier versions of the architecture.

- Three new instructions (CPS, SRS, RFE) to improve nested stack handling of different exceptions in a common mode. CPS can also be used to efficiently enable or disable the interrupt and imprecise abort masks, either within a mode, or while transitioning from a privileged mode to any other mode. See *New instructions to improve exception handling* on page A2-27 for a brief description.

—— **Note** ——

The normal vector at address `0x00000014` and the high vector at address `0xFFFF0014` are reserved for future expansion.

Table A2-3 Exception processing modes

Exception type	Mode	VEª	Normal address	High vector address
Reset	Supervisor		0x00000000	0xFFFF0000
Undefined instructions	Undefined		0x00000004	0xFFFF0004
Software interrupt (SWI)	Supervisor		0x00000008	0xFFFF0008
Prefetch Abort (instruction fetch memory abort)	Abort		0x0000000C	0xFFFF000C
Data Abort (data access memory abort)	Abort		0x00000010	0xFFFF0010

Table A2-3 Exception processing modes

Exception type	Mode	VE[a]	Normal address	High vector address
IRQ (interrupt)	IRQ	0	0x00000018	0xFFFF0018
		1	Implementation defined	
FIQ (fast interrupt)	FIQ	0	0x0000001C	0xFFFF001C
		1	Implementation defined	

a. VE = vectored interrupt enable (CP15 control); RAZ when not implemented.

When an exception occurs, the banked versions of R14 and the SPSR for the exception mode are used to save state as follows:

```
R14_<exception_mode> = return link
SPSR_<exception_mode> = CPSR
CPSR[4:0] = exception mode number
CPSR[5] = 0                          /* Execute in ARM state */
if <exception_mode> == Reset or FIQ then
    CPSR[6] = 1                      /* Disable fast interrupts */
/* else CPSR[6] is unchanged */
CPSR[7] = 1                          /* Disable normal interrupts */
if <exception_mode> != UNDEF or SWI then
    CPSR[8] = 1                      /* Disable imprecise aborts (v6 only) */
/* else CPSR[8] is unchanged */
PC = exception vector address
```

To return after handling the exception, the SPSR is moved into the CPSR, and R14 is moved to the PC. This can be done atomically in two ways:

- using a data-processing instruction with the S bit set, and the PC as the destination
- using the Load Multiple with Restore CPSR instruction, as described in *LDM (3)* on page A4-39.

The following sections show what happens automatically when the exception occurs, and also show the recommended data-processing instruction to use to return from each exception. This instruction is always a MOVS or SUBS instruction with the PC as its destination.

——— **Note** ———

When the recommended data-processing instruction is a SUBS and a Load Multiple with Restore CPSR instruction is used to return from the exception handler, the subtraction must still be performed. This is usually done at the start of the exception handler, before the return link is stored to memory.

For example, an interrupt handler that wishes to store its return link on the stack might use instructions of the following form at its entry point:

```
SUB    R14, R14, #4
STMFD  SP!, {<other_registers>, R14}
```

and return using the instruction:

```
LDMFD   SP!, {<other_registers>, PC}^
```

A2.6.1 Reset

When the Reset input is asserted on the processor, the ARM processor immediately stops execution of the current instruction. When Reset is de-asserted, the following actions are performed:

```
R14_svc   = UNPREDICTABLE value
SPSR_svc  = UNPREDICTABLE value
CPSR[4:0] = 0b10011                /* Enter Supervisor mode */
CPSR[5]   = 0                      /* Execute in ARM state */
CPSR[6]   = 1                      /* Disable fast interrupts */
CPSR[7]   = 1                      /* Disable normal interrupts */
CPSR[8]   = 1                      /* Disable Imprecise Aborts (v6 only) */
if high vectors configured then
    PC    = 0xFFFF0000
else
    PC    = 0x00000000
```

After Reset, the ARM processor begins execution at address `0x00000000` or `0xFFFF0000` in Supervisor mode with interrupts disabled.

—— **Note** ——

There is no architecturally defined way of returning from a Reset.

A2.6.2 Undefined Instruction exception

If the ARM processor executes a coprocessor instruction, it waits for any external coprocessor to acknowledge that it can execute the instruction. If no coprocessor responds, an Undefined Instruction exception occurs.

If an attempt is made to execute an instruction that is UNDEFINED, an Undefined Instruction exception occurs (see *Extending the instruction set* on page A3-35).

The Undefined Instruction exception can be used for software emulation of a coprocessor in a system that does not have the physical coprocessor (hardware), or for general-purpose instruction set extension by software emulation.

When an Undefined Instruction exception occurs, the following actions are performed:

```
R14_und   = address of next instruction after the undefined instruction
SPSR_und  = CPSR
CPSR[4:0] = 0b11011                /* Enter Undefined Instruction mode */
CPSR[5]   = 0                      /* Execute in ARM state */
/* CPSR[6] is unchanged */
CPSR[7]   = 1                      /* Disable normal interrupts */
/* CPSR[8] is unchanged */
if high vectors configured then
    PC    = 0xFFFF0004
else
    PC    = 0x00000004
```

To return after emulating the undefined instruction use:

```
    MOVS PC,R14
```

This restores the PC (from R14_und) and CPSR (from SPSR_und) and returns to the instruction following the undefined instruction.

In some coprocessor designs, an internal exceptional condition caused by one coprocessor instruction is signaled *imprecisely* by refusing to respond to a later coprocessor instruction. In these circumstances, the Undefined Instruction handler takes whatever action is necessary to clear the exceptional condition, then returns to the second coprocessor instruction. To do this use:

```
    SUBS PC,R14,#4
```

A2.6.3   Software Interrupt exception

The Software Interrupt instruction (SWI) enters Supervisor mode to request a particular supervisor (operating system) function. When a SWI is executed, the following actions are performed:

```
R14_svc   = address of next instruction after the SWI instruction
SPSR_svc  = CPSR
CPSR[4:0] = 0b10011          /* Enter Supervisor mode */
CPSR[5]   = 0                /* Execute in ARM state */
/* CPSR[6] is unchanged */
CPSR[7]   = 1                /* Disable normal interrupts */
/* CPSR[8] is unchanged */
if high vectors configured then
    PC    = 0xFFFF0008
else
    PC    = 0x00000008
```

To return after performing the SWI operation, use the following instruction to restore the PC (from R14_svc) and CPSR (from SPSR_svc) and return to the instruction following the SWI:

```
    MOVS PC,R14
```

A2.6.4 Prefetch Abort (instruction fetch memory abort)

A memory abort is signaled by the memory system. Activating an abort in response to an instruction fetch marks the fetched instruction as invalid. A Prefetch Abort exception is generated if the processor tries to execute the invalid instruction. If the instruction is not executed (for example, as a result of a branch being taken while it is in the pipeline), no Prefetch Abort occurs.

In ARMv5 and above, a Prefetch Abort exception can also be generated as the result of executing a BKPT instruction. For details, see *BKPT* on page A4-14 (ARM instruction) and *BKPT* on page A7-24 (Thumb instruction).

When an attempt is made to execute an aborted instruction, the following actions are performed:

```
R14_abt   = address of the aborted instruction + 4
SPSR_abt  = CPSR
CPSR[4:0] = 0b10111              /* Enter Abort mode */
CPSR[5]   = 0                    /* Execute in ARM state */
/* CPSR[6] is unchanged */
CPSR[7]   = 1                    /* Disable normal interrupts */
CPSR[8]   = 1                    /* Disable Imprecise Data Aborts (v6 only) */
if high vectors configured then
    PC    = 0xFFFF000C
else
    PC    = 0x0000000C
```

To return after fixing the reason for the abort, use:

```
    SUBS PC,R14,#4
```

This restores both the PC (from R14_abt) and CPSR (from SPSR_abt), and returns to the aborted instruction.

A2.6.5 Data Abort (data access memory abort)

A memory abort is signaled by the memory system. Activating an abort in response to a data access (load or store) marks the data as invalid. A Data Abort exception occurs before any following instructions or exceptions have altered the state of the CPU. The following actions are performed:

```
R14_abt   = address of the aborted instruction + 8
SPSR_abt  = CPSR
CPSR[4:0] = 0b10111              /* Enter Abort mode */
CPSR[5]   = 0                    /* Execute in ARM state */
/* CPSR[6] is unchanged */
CPSR[7]   = 1                    /* Disable normal interrupts */
CPSR[8]   = 1                    /* Disable Imprecise Data Aborts (v6 only) */
if high vectors configured then
    PC    = 0xFFFF0010
else
    PC    = 0x00000010
```

To return after fixing the reason for the abort use:

```
SUBS PC,R14,#8
```

This restores both the PC (from R14_abt) and CPSR (from SPSR_abt), and returns to re-execute the aborted instruction.

If the aborted instruction does not need to be re-executed use:

```
SUBS PC,R14,#4
```

Effects of data-aborted instructions

Instructions that access data memory can modify memory by storing one or more values. If a Data Abort occurs in such an instruction, the value of each memory location that the instruction stores to is:

- unchanged if the memory system does not permit write access to the memory location
- UNPREDICTABLE otherwise.

Instructions that access data memory can modify registers in three ways:

- By loading values into one or more of the general-purpose registers, that can include the PC.

- By specifying *base register write-back*, in which the base register used in the address calculation has a modified value written to it. All instructions that allow this to be specified have UNPREDICTABLE results if base register write-back is specified and the base register is the PC, so only general-purpose registers other than the PC can legitimately be modified in this way.

- By loading values into coprocessor registers.

If a Data Abort occurs, the values left in these registers are determined by the following rules:

1. The PC value on entry to the Data Abort handler is 0x00000010 or 0xFFFF0010, and the R14_abt value is determined from the address of the aborted instruction. Neither is affected in any way by the results of any PC load specified by the instruction.

2. If base register write-back is not specified, the base register value is unchanged. This applies even if the instruction loaded its own base register and the memory access to load the base register occurred earlier than the aborting access.

   For example, suppose the instruction is:

   ```
   LDMIA R0,{R0,R1,R2}
   ```

   and the implementation loads the new R0 value, then the new R1 value and finally the new R2 value. If a Data Abort occurs on any of the accesses, the value in the base register R0 of the instruction is unchanged. This applies even if it was the load of R1 or R2 that aborted, rather than the load of R0.

3. If base register write-back is specified, the value left in the base register is determined by the *abort model* of the implementation, as described in *Abort models* on page A2-22.

4. If the instruction only loads one general-purpose register, the value in that register is unchanged.

5. If the instruction loads more than one general-purpose register, UNPREDICTABLE values are left in destination registers that are neither the PC nor the base register of the instruction.

6. If the instruction loads coprocessor registers, UNPREDICTABLE values are left in the destination coprocessor registers, unless otherwise specified in the instruction set description of the specific coprocessor.

Abort models

The abort model used by an ARM implementation is IMPLEMENTATION DEFINED, and is one of the following:

Base Restored Abort Model

If a Data Abort occurs in an instruction that specifies base register write-back, the value in the base register is unchanged. This is the only abort model permitted in ARMv6 and above.

Base Updated Abort Model

If a Data Abort occurs in an instruction that specifies base register write-back, the base register write-back still occurs. This model is prohibited in ARMv6 and above.

In either case, the abort model applies uniformly across all instructions. An implementation does not use the Base Restored Abort Model for some instructions and the Base Updated Abort Model for others.

A2.6.6 Imprecise aborts

An imprecise data abort, caused, for example, by an External Error on a write that has been held in a Write Buffer, is asynchronous to the execution of the causing instruction and might in reality occur many cycles after the instruction that caused the memory access has retired. For this reason, the imprecise data abort might occur at a time that the processor is in abort mode because of a precise abort, or might have live state in abort mode, but be handling an interrupt.

To avoid the loss of the Abort mode state (R14 and SPSR_abt) in these cases, that would lead to the processor entering an unrecoverable state, the existence of a pending imprecise data abort must be held by the system until such time as the abort mode can safely be entered.

From ARMv6, a mask is added into the CPSR (CPSR[8]) to control when an imprecise abort cannot be accepted. This bit is referred to as the A bit. The imprecise data abort causes a Data Abort to be taken when imprecise data aborts are not masked. When imprecise data aborts are masked, the implementation is responsible for holding the presence of a pending imprecise abort until the mask is cleared and the abort is taken.

The A bit is set automatically on entry into Abort Mode, FIQ Mode, IRQ mode, and on reset.

The A bit can only be changed from a privileged mode.

A2.6.7 Interrupt request (IRQ) exception

The IRQ exception is generated externally by asserting the IRQ input on the processor. It has a lower priority than FIQ (see Table A2-4 on page A2-25), and is masked out when an FIQ sequence is entered.

Interrupts are disabled when the I bit in the CPSR is set. If the I bit is clear, ARM checks for an IRQ at instruction boundaries.

—— **Note** ——

The I bit can only be changed from a privileged mode.

When an IRQ is detected, the following actions are performed:

```
R14_irq  = address of next instruction to be executed + 4
SPSR_irq = CPSR
CPSR[4:0] = 0b10010              /* Enter IRQ mode */
CPSR[5]   = 0                    /* Execute in ARM state */
/* CPSR[6] is unchanged */
CPSR[7]   = 1                    /* Disable normal interrupts */
CPSR[8]   = 1                    /* Disable Imprecise Data Aborts (v6 only) */
if VE==0 then
  if high vectors configured then
      PC   = 0xFFFF0018
  else
      PC   = 0x00000018
else
  PC = IMPLEMENTATION DEFINED    /* see page A2-26 */
```

To return after servicing the interrupt, use:

```
    SUBS PC,R14,#4
```

This restores both the PC (from R14_irq) and CPSR (from SPSR_irq), and resumes execution of the interrupted code.

A2.6.8   Fast interrupt request (FIQ) exception

The FIQ exception is generated externally by asserting the FIQ input on the processor. FIQ is designed to support a data transfer or channel process, and has sufficient private registers to remove the need for register saving in such applications, therefore minimizing the overhead of context switching.

Fast interrupts are disabled when the F bit in the CPSR is set. If the F bit is clear, ARM checks for an FIQ at instruction boundaries.

——— **Note** ———

The F bit can only be changed from a privileged mode.

When an FIQ is detected, the following actions are performed:

```
R14_fiq  = address of next instruction to be executed + 4
SPSR_fiq = CPSR
CPSR[4:0] = 0b10001                /* Enter FIQ mode */
CPSR[5]  = 0                       /* Execute in ARM state */
CPSR[6]  = 1                       /* Disable fast interrupts */
CPSR[7]  = 1                       /* Disable normal interrupts */
CPSR[8]  = 1                       /* Disable Imprecise Data Aborts (v6 only) */
if VE==0 then
  if high vectors configured then
      PC    = 0xFFFF001C
  else
      PC    = 0x0000001C
else
  PC = IMPLEMENTATION DEFINED    /* see page A2-26 */
```

To return after servicing the interrupt, use:

```
    SUBS PC, R14,#4
```

This restores both the PC (from R14_fiq) and CPSR (from SPSR_fiq), and resumes execution of the interrupted code.

The FIQ vector is deliberately the last vector to allow the FIQ exception-handler software to be placed directly at address `0x0000001C` or `0xFFFF001C`, without requiring a branch instruction from the vector.

A2.6.9   Exception priorities

Table A2-4 shows the exception priorities:

Table A2-4 Exception priorities

Priority		Exception
Highest	1	Reset
	2	Data Abort (including data TLB miss)
	3	FIQ
	4	IRQ
	5	Imprecise Abort (external abort) - ARMv6
	6	Prefetch Abort (including prefetch TLB miss)
Lowest	7	Undefined instruction SWI

Undefined instruction and software interrupt cannot occur at the same time, because they each correspond to particular (non-overlapping) decodings of the current instruction. Both must be lower priority than Prefetch Abort, because a Prefetch Abort indicates that no valid instruction was fetched.

The priority of a Data Abort exception is higher than FIQ, which ensures that the Data Abort handler is entered before the FIQ handler is entered (so that the Data Abort is resolved after the FIQ handler has completed).

A2.6.10   High vectors

High vectors were introduced into some implementations of ARMv4 and are required in ARMv6 implementations. High vectors allow the exception vector locations to be moved from their normal address range 0x00000000-0x0000001C at the bottom of the 32-bit address space, to an alternative address range 0xFFFF0000-0xFFFF001C near the top of the address space. These alternative locations are known as the *high vectors*.

Prior to ARMv6, it is IMPLEMENTATION DEFINED whether the high vectors are supported. When they are, a hardware configuration input selects whether the normal vectors or the high vectors are to be used from reset.

The ARM instruction set does not contain any instructions that can directly change whether normal or high vectors are configured. However, if the standard System Control coprocessor is attached to an ARM processor that supports the high vectors, bit[13] of coprocessor 15 register 1 can be used to switch between using the normal vectors and the high vectors (see *Register 1: Control registers* on page B3-13).

A2.6.11 Vectored interrupt support

Historically, the IRQ and FIQ exception vectors are affected by whether high vectors are enabled, and are otherwise fixed. The result is that interrupt handlers typically have to start with an instruction sequence to determine the cause of the interrupt and branch to a routine to handle it. Support of vectored interrupts allows an interrupt controller to prioritize interrupts, and provide the required interrupt handler address directly to the core. The vectored interrupt behavior is explicitly enabled by the setting of a bit, the VE bit, in the system coprocessor CP15 register 1. See *Register 1: Control registers* on page B3-13. For backwards compatibility, the vectored interrupt mechanism is disabled on reset. The details of the hardware to support vectored interrupts is IMPLEMENTATION DEFINED.

A vectored interrupt controller (VIC) can reduce effective interrupt latency considerably, by eliminating the need for an interrupt handler to identify the source of an interrupt and acknowledge it before re-enabling the interrupts. Furthermore, if the VIC and core implement an appropriate handshake as the interrupt handler routine is entered, the VIC can automatically mask out the interrupt source associated with that handler and any lower priority sources. This allows the interrupts concerned to be re-enabled by the processor core as soon as their return information (that is, R14 and SPSR values) have been saved, reducing the period during which higher priority interrupts are disabled.

A2.6.12 Low interrupt latency configuration

The FI bit (bit[21]) in the system control register (CP15 register 1) enables the interrupt latency configuration logic in an implementation. See *Register 1: Control registers* on page B3-13. The purpose of this configuration is to reduce the interrupt latency of the processor. The exact mechanisms that are used to perform this are IMPLEMENTATION DEFINED.

In order to ensure that a change between normal and low interrupt latency configurations is synchronized correctly, the FI bit must only be changed in IMPLEMENTATION DEFINED circumstances. It is recommended that software systems should only change the FI bit shortly after reset, while interrupts are disabled.

When interrupt latency is reduced, this may result in reduced performance overall. Examples of the mechanisms which may be used are disabling Hit-Under-Miss functionality within a core, and the abandoning of restartable external accesses, allowing the core to react to a pending interrupt faster than would otherwise be the case. Low interrupt latency configuration may have IMPLEMENTATION DEFINED effects in the memory system or elsewhere outside the processor core.

In low interrupt latency configuration, software must only use multi-word load/store instructions in ways that are fully restartable. This allows (but does not require) implementations to make multi-word instructions interruptible when in low interrupt latency configuration. The multi-access instructions to which this rule currently applies are:

ARM   LDC, all forms of LDM, LDRD, STC, all forms of STM, STRD

Thumb   LDMIA, PUSH, POP, STMIA

——— **Note** ———

If the instruction is interrupted before it is complete, the result may be that one or more of the words are accessed twice. Idempotent memory (multiple reads or writes of the same information exhibit identical system results) is a requirement of system correctness.

In ARMv6, memory with the normal attribute is guaranteed to behave this way, however, memory marked as Device or Strongly Ordered is not (for example, a FIFO). It is IMPLEMENTATION DEFINED whether multi-word accesses are supported for Device and Strongly Ordered memory types in the low interrupt latency configuration.

A similar situation exists with regard to multi-word load/store instructions that access memory locations that can abort in a recoverable way, since an abort on one of the words accessed may cause a previously-accessed word to be accessed twice – once before the abort, and a second time after the abort handler has returned. The requirement in this case is either that all side-effects are idempotent, or that the abort must either occur on the first word accessed or not at all.

A2.6.13 New instructions to improve exception handling

ARMv6 adds an instruction to simplify changes of processor mode and the disabling and enabling of interrupts. New instructions are also added to reduce the processing cost of handling exceptions in a different mode to the exception entry mode, by removing any need to use the original mode's stack. Two examples are:

*   IRQ routines may wish to execute in System or Supervisor mode, so that they can both re-enable IRQs and use BL instructions. This is not possible in IRQ mode, because a nested IRQ could corrupt the BL's return link at any time. Using the new instructions, the system can store the return state (R14 link register and SPSR_irq) to the System/User or Supervisor mode stack, switch to System or Supervisor mode and re-enable IRQs efficiently, without making any use of R13_irq or the IRQ stack.

*   FIQ mode is designed for efficient use by a single owner, using R8_fiq – R13_fiq as global variables. In addition, unlike IRQs, FIQs are not disabled by other exceptions (apart from reset), making them the preferred type for real time interrupts, when other exceptions are being used routinely, such as virtual memory or instruction emulation. IRQs may be disabled for unacceptably long periods of time while these needs are being serviced.

    However, if more than one real-time interrupt source is required, there is a conflict of interest. The new mechanism allows multiple FIQ sources and minimizes the period with FIQs disabled, greatly reducing the interrupt latency penalty. The FIQ mode registers can be allocated to the highest priority FIQ as a single owner.

SRS – Store Return State

This instruction stores R14_<current_mode> and SPSR_<current_mode> to sequential addresses, using the banked version of R13 for a specified mode to supply the base address (and to be written back to if base register writeback is specified). This allows an exception handler to store its return state on a stack other than the one automatically selected by its exception entry sequence.

The addressing mode used is a version of ARM addressing mode 4 (see *Addressing Mode 4 - Load and Store Multiple* on page A5-52), modified so as to assume a {R14,SPSR} register list rather than using a list specified by a bit mask in the instruction. This allows the SRS instruction to access stacks in a manner compatible with the normal use of STM instructions for stack accesses. See *SRS* on page A4-182 for the instruction details.

RFE – Return From Exception

This instruction loads the PC and CPSR from sequential addresses. This is used to return from an exception which has had its return state saved using the SRS instruction, and again uses a version of ARM addressing mode 4, modified this time to assume a {PC,CPSR} register list. See *RFE* on page A4-114 for the instruction details.

CPS – Change Processor State

This instruction provides new values for the CPSR interrupt masks, mode bits, or both, and is designed to shorten and speed up the read/modify/write instruction sequence used in earlier architecture variants to perform such tasks. Together with the SRS instruction, it allows an exception handler to save its return information on another mode's stack and then switch to that other mode, without modifying the stack belonging to the original mode or any registers other than the new mode's stack pointer.

The instruction also streamlines interrupt mask handling and mode switches in other code, and in particular allows short, efficient, atomic code sequences in a uniprocessor system by disabling interrupts at their start and re-enabling interrupts at their end. See *CPS* on page A4-28 for the instruction details.

A CPS Thumb instruction that allows mask updates within the current mode is also provided, see section *CPS* on page A7-39.

——— **Note** ———

The Thumb instruction cannot change the mode due to instruction space usage constraints.

A2.7   Endian Support

This section discusses memory and memory-mapped I/O, with regard to the assumptions ARM processor implementations make about endianness.

ARMv6 introduces several architectural extensions to support mixed-endian access in hardware:

*   Byte reverse instructions that operate on general-purpose register contents to support word, and signed and unsigned halfword data quantities.

*   Separate instruction and data endianness, with instructions fixed as little-endian format, naturally aligned, but with legacy support for 32-bit word-invariant binary images/ROM.

*   A PSR Endian control flag, the E bit, cleared on reset and exception entry. This adds a byte-reverse operation to the entire load and store instruction space when data is loaded into, and stored back out of the register file. In previous architectures this PSR bit was specified as 0 and are never set in legacy code written to conform to architectures prior to ARMv6.

*   ARM and Thumb instructions to set and clear the E bit explicitly.

*   A byte-invariant addressing scheme to support fine-grain big-endian and little-endian shared data structures, to conform to the *IEEE Standard for Shared-Data Formats Optimized for Scalable Coherent Interface (SCI) Processors*, IEEE Std 1596.5-1993 (ISBN 1-55937-354-7, IEEE).

*   Bus interface endianness is IMPLEMENTATION DEFINED. However, it must support byte lane controls for unaligned word and halfword data access.

ARMv6 provides legacy behavior. For backwards compatibility, ARMv6 systems always reset to legacy mode.

A2.7.1   Address space

The ARM architecture uses a single, flat address space of 2^{32} 8-bit bytes. Byte addresses are treated as unsigned numbers, running from 0 to $2^{32} - 1$.

This address space is regarded as consisting of 2^{30} 32-bit words, each of whose addresses is word-aligned, which means that the address is divisible by 4. The word whose word-aligned address is A consists of the four bytes with addresses A, A+1, A+2 and A+3.

In ARM architecture version 4 and above, the address space is also regarded as consisting of 2^{31} 16-bit halfwords, each of whose addresses is halfword-aligned (divisible by 2). The halfword whose halfword-aligned address is A consists of the two bytes with addresses A and A+1.

In ARM architecture version 5E and above, the address space supports 64-bit doubleword operations. Doubleword operations can be considered as two-word load/store operations, each word addressed as follows:
*   A, A+1, A+2, and A+3 for the first word
*   A+4, A+5, A+6, and A+7 for the second word.

Prior to version 6, doubleword operations could be word-aligned (IMPLEMENTATION DEPENDENT) with doubleword-aligned addresses always supported. (the address is divisible by 8). Version 6 mandates support of both modulo4 and modulo8 alignment of doublewords, and introduces support for unaligned word and halfword data accesses, all controlled through the standard System Control coprocessor.

Java state (see *The J bit* on page A2-12) introduced with ARM architecture variant v5J supports byte addressing.

Address calculations are normally performed using ordinary integer instructions. This means that they normally wrap around if they overflow or underflow the address space. This means that the result of the calculation is reduced modulo 2^{32}. However, to minimize the chances of incompatibility if the address space is extended in the future, programs should not be written so that they rely on this behavior. Address calculations should be written so that their results would still lie in the range 0 to $2^{32} – 1$ if they were calculated without wrap-around.

Most branch instructions calculate their targets by adding an instruction-specified offset to the value of the PC and writing the result back to the PC. If the overall effect of this calculation of:

```
(address_of_current_instruction) + 8 + offset
```

is to overflow or underflow the address space, the instruction is technically UNPREDICTABLE because it relies on address wrap-around. The result of this is that forward branches past address `0xFFFFFFFF` and backward branches past address `0x00000000` should not be used.

Also, normal sequential execution of instructions effectively calculates:

```
(address_of_current_instruction) + 4
```

after each instruction to determine which instruction to execute next. If this calculation overflows the top of the address space, the result is again technically UNPREDICTABLE. In other words, programs should not rely on sequential execution of the instruction at address `0x00000000` after the instruction at address `0xFFFFFFFC`.

The above only applies to instructions that are executed, including those which fail their condition code check. Most ARM implementations prefetch instructions ahead of the currently-executing instruction. If this prefetching overflows the top of the address space, it does not cause the implementation's behavior to become UNPREDICTABLE until and unless the prefetched instructions are actually executed.

LDC, LDM, STC, and STM instructions access a sequence of words at increasing memory addresses, effectively incrementing a memory address by 4 for each load or store. If this calculation overflows the top of the address space, the result is again technically UNPREDICTABLE. In other words, programs should not use these instructions in such a way that they access the word at address `0x00000000` sequentially after the word at address `0xFFFFFFFC`.

A2.7.2   Endianness - an overview

The rules in *Address space* on page A2-29 require that for a word-aligned address A:

- the word at address A consists of the bytes at addresses A, A+1, A+2 and A+3

- the halfword at address A consists of the bytes at addresses A and A+1

- the halfword at address A+2 consists of the bytes at addresses A+2 and A+3.

- the word at address A therefore consists of the halfwords at addresses A and A+2.

However, this does not totally specify the mappings between words, halfwords, and bytes.

A memory system uses one of the two following mapping schemes. This choice is known as the endianness of the memory system.

In a *little-endian* memory system:

- a byte or halfword at a word-aligned address is the least significant byte or halfword within the word at that address

- a byte at a halfword-aligned address is the least significant byte within the halfword at that address.

In a *big-endian* memory system:

- a byte or halfword at a word-aligned address is the most significant byte or halfword within the word at that address

- a byte at a halfword-aligned address is the most significant byte within the halfword at that address.

For a word-aligned address A, tables Table A2-5 Table A2-6 show how the word at address A, the halfwords at addresses A and A+2, and the bytes at addresses A, A+1, A+2 and A+3 map on to each other for each endianness.

Table A2-5 Big-endian memory system

3 1		2 2 4 3		1 1 6 5		8 7		0
Word at Address A								
Halfword at Address A				Halfword at Address A+2				
Byte at Address A		Byte at Address A+1		Byte at Address A+2		Byte at Address A+3		

Table A2-6 Little-endian memory system

3 1		2 2 4 3		1 1 6 5		8 7		0
Word at Address A								
Halfword at Address A+2				Halfword at Address A				
Byte at Address A+3		Byte at Address A+2		Byte at Address A+1		Byte at Address A		

On memory systems wider than 32 bits, the ARM architecture has traditionally supported a word-invariant memory model, meaning that a word aligned address will fetch the same data in both big endian and little endian systems. This is illustrated for a 64-bit data path in Table A2-7 and Table A2-8.

Table A2-7 Big-endian word invariant case

6 3		3 2	3 1		0
Word at Address A+2			Word at Address A		
Halfword at Address A+4	Halfword at Address A+6		Halfword at Address A	Halfword at Address A+2	

Table A2-8 Little-endian word invariant case

6 3		3 2	3 1		0
Word at Address A+2			Word at Address A		
Halfword at Address A+6	Halfword at Address A+4		Halfword at Address A+2	Halfword at Address A	

New provisions in ARMv6

ARMv6 has introduced new configurations known as mixed endian support. These use a byte-invariant address model, affecting the order that bytes are transferred to and from ARM registers. Byte invariance means that the address of a byte in memory is the same irrespective if that byte is being accessed in a big endian or little endian manner as illustrated in *Byte invariant data access* on page A2-36.

Byte, halfword, and word accesses access the same one, two or four bytes in memory for both big and little endian configuration. Double word and multiple word accesses in the ARM architecture are treated as a series of word accesses from incrementing word addresses, and hence each word also returns the same bytes of information in these cases too.

——— **Note** ———

When an implementation is configured in mixed endian mode, this only affects data accesses and how they are loaded/stored to/from the register file. Instruction fetches always assume a little endian byte order model.

• When configured for big endian load/store, the lowest address provides the most significant byte of the requested word or halfword. For LDRD/STRD this is the most significant byte of the first word accessed.

• When configured for little endian load/store, the lowest address provides the least significant byte of the requested word or halfword. For LDRD/STRD this is the least significant byte of the first word accessed.

The convention adopted in this book is to identify the different endian models as follows:

• the word invariant big endian model is known as BE-32

• the byte invariant big endian model is referred to as BE-8

• little endian data is identical in both models and referred to as LE.

A2.7.3   Endian configuration and control

Prior to ARMv6, a single bit (B-bit) provides endian control. It is IMPLEMENTATION DEFINED whether implementations of ARM architecture version 5 and below support little-endian memory systems, big-endian memory systems, or both. If a standard System Control coprocessor is attached to an ARM implementation supporting the B-bit, this configuration input can be changed by writing to bit[7] of register 1 of the System Control coprocessor (see *Register 1: Control registers* on page B3-13). An implementation may preset the B-bit on reset. If an ARM processor configures for little-endian operation on reset, and it is attached to a big-endian memory system, one of the first things the reset handler must do is switch the configured endianness to big-endian, using an instruction sequence like:

```
MRC    p15, 0, r0, c1, c0    ; r0 := CP15 register 1
ORR    r0, r0, #0x80         ; Set bit[7] in r0
MCR    p15, 0, r0, c1, c0    ; CP15 register 1 := r0
```

This must be done before there is any possibility of a byte or halfword data access occurring, or instruction execution in Thumb or Java state.

ARMv6 supports big-endian, little-endian, and byte-invariant hybrid systems. Features are provided in the System Control coprocessor and CPSR/SPSR to support hybrid operation. The System Control Coprocessor register (CP15 register 1) and CPSR bits used are:

• Bit[1] - A-bit - used to enable alignment checking. Always reset to zero (alignment checking OFF).

• Bit[7] - B-bit - retained for backwards compatibility

• Bit[22] - the U-bit - enables v6 unaligned data support, and used with Bit[1] - the A-bit - to determine alignment checking behavior.

• Bit [25] - the EE-bit - Exception Endian bit.

• CPSR/SPSR[9] - the E-bit - load/store endian control.

The behavior of the memory system with respect to the U and A bits is summarized in Table A2-9.

Table A2-9

U	A	Description
0	0	Legacy (32-bit word invariant only)
0	1	Modulo 8 alignment checking: LDRD/STRD (8 and 32-bit invariant memory models)
1	0	Unaligned access support (8-bit byte invariant data accesses only)
1	1	Modulo 4 alignment checking: LDRD/STRD (8-bit and 32-bit invariant memory models)

The EE-bit value is used to overwrite the CPSR_E-bit on exception entry and for page table lookups. These are asynchronous events with respect to normal control of the CPSR E-bit.

A 2-bit configuration (CFGEND[1:0]) replaces the BigEndinit configuration pin to provide hardware system configuration on reset. CFGEND[1] maps to the U-bit, while CFGEND[0] sets either the B-bit or EE-bit and CPSR_E on reset.

Table A2-10 defines the CFGEND[1:0] encoding and associated configurations.

Table A2-10

CPGEND[1:0]	Coprocessor 15 System Control Register (register 1)				CPSR/SPSR
	EE-bit[25]	U-bit[22]	A-bit[1]	B-bit[7]	E-bit
00	0	0	0	0	0
01	0	0	0	1	0
10	0	1	0	0	0
11	1	1	0	0	1

Where an implementation does not include configuration pins, the U-bit and A-bit shall clear on reset.

The usage model for the U-bit and A-bit with respect to the B-bit and E-bit is summarized in Table A2-11. Interaction of these control bits with data alignment is discussed in section *Unaligned access support* on page A2-42.

Table A2-11 Endian and Alignment Control Bit Usage Summary

U	A	B	E	Instruction Endianness	Data Endianness	Unaligned Behavior	Description
0	0	0	0	LE	LE	Rotated LDR	Legacy LE / programmed BE configuration
0	0	0	1	-	-	-	RESERVED (no E-bit in legacy code)
0	0	1	0	BE-32	BE-32	Rotated LDR	Legacy BE (32-bit word-invariant)
0	0	1	1	-	-	-	RESERVED (no E-bit in legacy code)
0	1	0	0	LE	LE	Data Abort	modulo 8 LDRD/STRD doubleword alignment checking. LE Data
0	1	0	1	LE	BE-8	Data Abort	modulo 8 LDRD/STRD doubleword alignment checking. BE Data
0	1	1	0	BE-32	BE-32	Data Abort	modulo 8 LDRD/STRD doubleword alignment checking.
0	1	1	1	-	-	-	RESERVED
1	0	0	0	LE	LE	Unaligned	LE instructions, LE mixed-endian data, unaligned access permitted
1	0	0	1	LE	BE-8	Unaligned	LE instructions, BE mixed-endian data, unaligned access permitted
1	0	1	x	-	-	-	RESERVED
1	1	0	0	LE	LE	Data Abort	LE OS, strict V6 alignment checking, LE Data

Table A2-11 Endian and Alignment Control Bit Usage Summary

U	A	B	E	Instruction Endianness	Data Endianness	Unaligned Behavior	Description
1	1	0	1	LE	BE-8	Data Abort	LE OS, strict V6 alignment checking, BE data
1	1	1	0	BE-32	BE-32	Data Abort	BE OS, strict V6 alignment checking
1	1	1	1	-	-	-	RESERVED

BE-32 and BE-8 are as defined in section *Endianness - an overview* on page A2-30. Data aborts cause an alignment error to be reported in the Fault Status Register in the system coprocessor.

———— **Note** ————

The U, A and B bits are System Control Coprocessor bits, while the E bit is a CPSR/SPSR flag.

The behavior of SETEND instructions (or any other instruction that modifies the CPSR) is UNPREDICTABLE when setting the E-bit would result in a RESERVED state.

A2.7.4 Byte invariant data access

The architectural specification of unaligned data representations is defined in terms of the bytes transferred between memory and register, regardless of bus width and bus endianness.

Little-endian data items are described using lowercase byte labelling bX..b0 (byteX to byte 0), and a pointer is always treated as pointing to the least significant byte of the addressed data.

Big-endian data items are described using uppercase byte labelling B0..BX (BYTE0 to BYTEX), and a pointer is always treated as pointing to the most significant byte of the addressed data.

Load unsigned Byte, endian independent

The addressed byte is loaded from memory into bits[7:0] of the general-purpose register and bits[31:8] are zeroed, as shown in Figure A2-2.
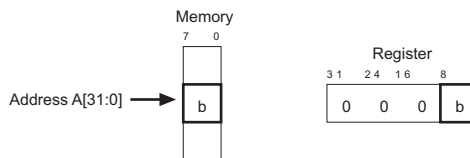
Figure A2-2 LDRB

Load Signed Byte, endian independent

The addressed byte is loaded from the memory into bits[7:0] of the general-purpose register and the sign bit is extended into bits[31:8] of the register, as shown in Figure A2-3.
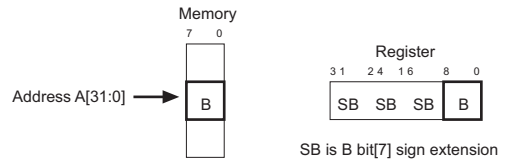
Figure A2-3 LDRSB

Store Byte, endian independent

Bits[7:0] of the general-purpose register are stored into the addressed byte in memory, as shown in Figure A2-4.
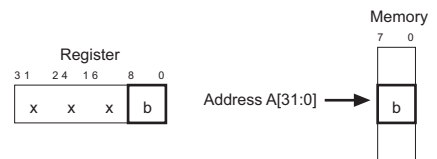
Figure A2-4 STRB

Load unsigned Halfword, little-endian

The addressed byte-pair is loaded from memory into bits[15:0] of the general-purpose register, bits[31:8] are zeroed such that the least significant (addressed) byte in memory appears in bits[7:0] of the ARM register, as shown in Figure A2-5.
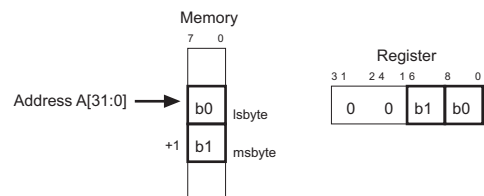
Figure A2-5 LDRH, E = 0

Load unsigned Halfword, big-endian

The addressed byte-pair is loaded from memory into bits[15:0] of the general-purpose register. Bits[31:8] are zeroed such that the most significant (addressed) byte in memory appears in bits[15:8] of the ARM register, as shown in Figure A2-6 on page A2-38.

Memory

7   0

Address A[31:0] → B0   msbyte

+1   B1   lsbyte

Register

3 1   2 4 1 6   8   0

0   0   B0 B1
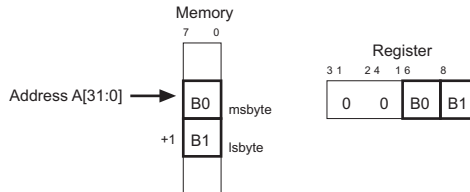
Figure A2-6 LDRH, E = 1

Load Signed Halfword, little-endian

The addressed byte-pair is loaded from memory into bits[15:0] of the general-purpose register, such that the least significant (addressed) byte in memory appears in bits[7:0] of the ARM register and bits[31:8] are sign-extended from bit[15], as shown in Figure A2-7.

Memory

7   0

Address A[31:0] → b0   lsbyte

+1   b1   msbyte

Register

3 1   2 4 1 6   8   0

sb1 sb1   b1   b0

sb1 is b1 bit[7] sign extension
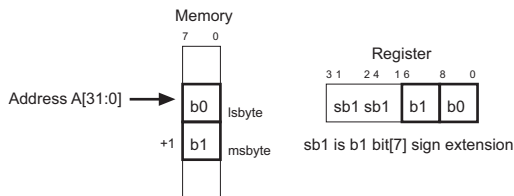
Figure A2-7 LDRSH, E = 0

Load Signed Halfword, big-endian

The addressed byte-pair is loaded from memory into bits[15:0] of the general-purpose register, such that the most significant (addressed) byte in memory appears in bits[15:8] of the ARM register and bits[31:16] replicate the sign-bit in bit[15], as shown in Figure A2-8.

Memory

7   0

Address A[31:0] → B0   msbyte

+1   B1   lsbyte

Register

3 1   2 4 1 6   8   0

SB0 SB0   B0   B1

SB0 is B0 bit[7] sign extension
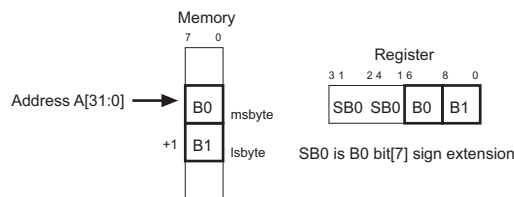
Figure A2-8 LDRSH, E = 1

Store Halfword, little-endian

Bits[15:0] of the general-purpose register are stored into the memory with bits [7:0] written to the addressed byte in memory and bits[15:8] to the next higher byte address in memory, as shown in Figure A2-9 on page A2-39.

Register

31   24 16   8   0

| x | x | b1 | b0 |

Memory

7   0

Address A[31:0] → | b0 | lsbyte

+1 | b1 | msbyte

Figure A2-9 STRH, E = 0

Store Halfword, big-endian

Bits[15:0] of the general-purpose register are stored into the memory with bits [15:8] written to the addressed byte in memory, bits[7:0] to the next higher byte address in memory, as shown in Figure A2-10.

Register

31   24 16   8   0

| x | x | B0 | B1 |

Memory

7   0

Address A[31:0] → | B0 | msbyte

+1 | B1 | lsbyte

Figure A2-10 STRH, E = 1

Load word, little-endian

The addressed byte-quad is loaded from memory into the 32-bit general-purpose register such that the least significant addressed byte in memory appears in bits[7:0] of the ARM register, as shown in Figure A2-11.

Memory

7   0

Address A[31:0] → | b0 | lsbyte

+1 | b1 |

+2 | b2 |

+3 | b3 | msbyte

Register

31   24 16   8   0

| b3 | b2 | b1 | b0 |
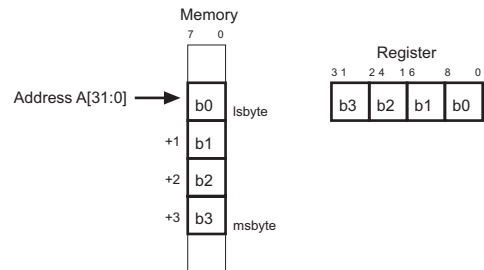
Figure A2-11 LDR, E = 0

Load word, big-endian

The addressed byte-quad is loaded from memory into the 32-bit general-purpose register such that the most significant addressed byte in memory appears in bits[31:24] of the ARM register, as shown in Figure A2-12 on page A2-40.

Memory

7   0

Address A[31:0] ——▶ | B0 | lsbyte

Register

3 1   2 4 1 6   8   0

| B0 | B1 | B2 | B3 |

+1 | B1 |

+2 | B2 |

+3 | B3 | msbyte
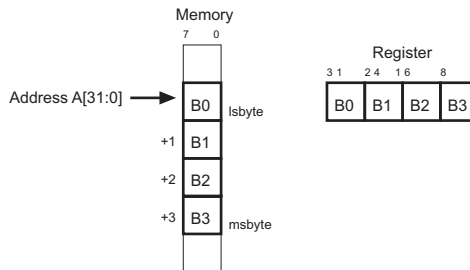
Figure A2-12 LDR, E = 1

Store word, little-endian

The 32-bit general-purpose register is stored to four bytes in memory where bits[7:0] of the ARM register are transferred to the least significant addressed byte in memory, as shown in Figure A2-13.

Register

3 1   2 4 1 6   8   0

| b3 | b2 | b1 | b0 |

Memory

7   0

Address A[31:0] ——▶ | b0 | lsbyte

+1 | b1 |

+2 | b2 |

+3 | b3 | msbyte
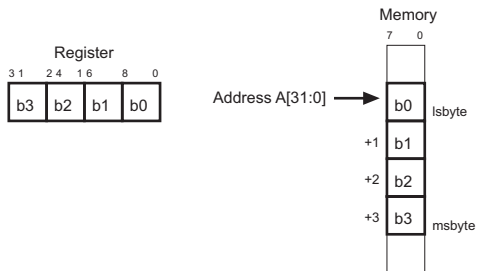
Figure A2-13 STR, E = 0

Store word, big-endian

The 32-bit general-purpose register is stored to four bytes in memory where bits[31:24] of the ARM register are transferred to the most significant addressed byte in memory, as shown in Figure A2-14.

Register

3 1   2 4 1 6   8   0

| B0 | B1 | B2 | B3 |

Memory

7   0

Address A[31:0] ——▶ | B0 | lsbyte

+1 | B1 |

+2 | B2 |

+3 | B3 | msbyte
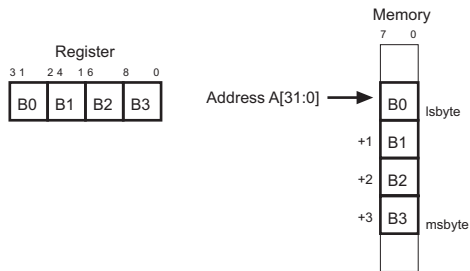
Figure A2-14 STR, E = 1

A2.7.5 Instructions to change CPSR E-bit

ARM and Thumb instructions are provided to set and clear the E-bit efficiently:

SETEND BE   Set the CPSR E bit.

SETEND LE   Reset the CPSR E bit.

These are unconditional instructions. See ARM *SETEND* on page A4-136 and Thumb *SETEND* on page A7-93.

A2.7.6 Instructions to reverse bytes in a general-purpose register

When an application or device driver has to interface to memory-mapped peripheral registers or shared-memory DMA structures that are not the same endianness as that of the internal data structures, or the endianness of the Operating System, an efficient way of being able to explicitly transform the endianness of the data is required.

ARMv6 ARM and Thumb instruction sets provide this functionality:

* Reverse word (four bytes) register, for transforming big and little-endian 32-bit representations. See ARM *REV* on page A4-110 and Thumb *REV* on page A7-86.

* Reverse halfword and sign-extend, for transforming signed 16-bit representations. See ARM *REVSH* on page A4-112 and Thumb *REVSH* on page A7-88.

* Reverse packed halfwords in a register for transforming big- and little-endian 16-bit representations. See ARM *REV16* on page A4-111 and Thumb *REV16* on page A7-87.

A2.8 Unaligned access support

The ARM architecture traditionally expects all memory accesses to be suitably aligned. In particular, the address used for a halfword access should normally be halfword-aligned, the address used for a word access should normally be word-aligned. Prior to version 6, it is IMPLEMENTATION DEPENDENT whether the address used for a doubleword access (LDRD/STRD) should be doubleword-aligned or word-aligned.

Memory accesses which are not aligned are called unaligned memory accesses and may generate an alignment fault or be UNPREDICTABLE.

For architecture variants prior to ARMv6, data access to non-aligned word and halfword data are treated as aligned from the memory interface perspective. That is:

• the address is treated as truncated, with address bits[1:0] treated as zero for word accesses, and address bit[0] treated as zero for halfword accesses.

• load single word ARM instructions are architecturally defined to rotate right the word-aligned data transferred by a non word-aligned address one, two or three bytes depending on the value of the two least significant address bits.

• alignment checking is defined for implementations supporting a System Control coprocessor using the A-bit in CP15 register 1. When this bit is set, a Data Abort indicating an alignment fault is reported for unaligned accesses.

ARMv6 introduces unaligned word and halfword load and store data access support. When this is enabled, the processor uses one or more memory accesses to generate the required transfer of adjacent bytes transparently to the programmer, apart from a potential access time penalty where the transaction crosses an IMPLEMENTATION DEFINED cache-line, bus-width or page boundary condition.

A2.8.1 Unaligned instruction fetches

All instruction fetches must be aligned. Specifically they must be:
• word aligned in ARM state
• halfword aligned in Thumb state.

Writing an unaligned address to R15 is UNPREDICTABLE, except in the specific cases where the instructions are associated with a Thumb to ARM state transition, bit[1] providing a valid address bit on transition to Thumb state, and bit[0] indicating whether a transition needs to occur. The BX instruction in ARM state (see *BX* on page A4-19) and POP instruction in Thumb state (see *POP* on page A7-81) are examples of instructions providing state transition support.

The general rules for reading and writing the program counter are defined in *The program counter, R15* on page A2-8.

A2.8.2 Unaligned data access in ARMv6 systems

ARMv6 uses the U-bit (CP15 register1 bit[22]) and A-bit (CP15 register1 bit[1]), to provide a configuration supporting the following unaligned memory accesses:

• Unaligned halfword accesses for LDRH, LDRSH and STRH.

- Unaligned word accesses for `LDR`, `LDRT`, `STR` and `STRT`.

The U-bit and A-bit are also used to configure endian support as described in *Endian configuration and control* on page A2-33. All other multi-byte load and store accesses shall be word aligned.

Instructions must always be aligned (and in little endian format):

- ARM instructions must be word-aligned
- Thumb instructions must be halfword-aligned.

In addition, an ARMv6 system shall reset to the CFGEND[1:0] condition as described in Table A2-10 on page A2-34.

For ARMv6, Table Table A2-13 on page A2-44 defines when an alignment fault must occur for an access, and when the behavior of an access is architecturally UNPREDICTABLE. It also gives details of precisely which memory locations are returned for valid accesses.

The access type descriptions used in this section are determined from the load/store instructions as describe in Table A2-12:

Table A2-12

Access Type	ARM instructions	Thumb instructions
Byte	`LDRB LDRBT LDRSB STRB STRBT SWPB` (either access)	`LDRB LDRSB STRB`
Halfword	`LDRH LDRSH STRH`	`LDRH LDRSH STRH`
WLoad	`LDR LDRT SWP` (load access, if U == 0)	`LDR`
WStore	`STR STRT SWP` (store access, if U == 0)	`STR`
WSync	`LDREX STREX SWP` (either access, if U == 1)	-
Two-word	`LDRD STRD`	-
Multi-word	`LDC LDM RFE SRS STC STM`	`LDMIA POP PUSH STMIA`

The following terminology is used to describe the memory locations accessed:

Byte[X] Means the byte whose address is X in the current endianness model. The correspondence between the endianness models is that Byte[A] in the LE endianness model, Byte[A] in the BE-8 endianness model, and Byte[A EOR 3] in the BE-32 endianness model are the same actual byte of memory.

Halfword[X] Means the halfword consisting of the bytes whose addresses are X and X+1 in the current endianness model, combined to form a halfword in little-endian order in the LE endianness model or in big-endian order in the BE-8 or BE-32 endianness model.

Word[X]   Means the word consisting of the bytes whose addresses are X, X+1, X+2, and X+3 in the current endianness model, combined to form a word in little-endian order in the LE endianness model or in big-endian order in the BE-8 or BE-32 endianness model.

────── **Note** ──────

It is a consequence of these definitions that if X is word-aligned, Word[X] consists of the same four bytes of actual memory in the same order in the LE and BE-32 endianness models.

──────────────

Align[X]   Means (X AND 0xFFFFFFFC) - that is, X with its least significant two bits forced to zero to make it word-aligned.

────── **Note** ──────

There is no difference between Addr and Align(Addr) on lines for which Addr[1:0] == 0b00 anyway. This can be exploited by implementations to simplify the control of when the least significant bits are forced to zero.

──────────────

For the Two-word and Multi-word access types, the Memory accessed column only specifies the lowest word accessed. Subsequent words have addresses constructed by successively incrementing the address of the lowest word by 4, and are constructed using the same endianness model as the lowest word.

Table A2-13 Data Access Behavior in ARMv6 Systems

U	A	Addr[2:0]	Access Types	Behavior	Memory accessed	Notes
0	0					LEGACY, NO ALIGNMENT FAULTING
0	0	xxx	Byte	Normal	Byte[Addr]	-
0	0	xx0	Halfword	Normal	Halfword[Addr]	-
0	0	xx1	Halfword	UNPREDICTABLE	-	-
0	0	xxx	WLoad	Normal	Word[Align(Addr)]	Loaded data rotated right by 8 * Addr[1:0] bits
0	0	xxx	WStore	Normal	Word[Align(Addr)]	Operation unaffected by Addr[1:0]
0	0	x00	WSync	Normal	Word[Addr]	-
0	0	xx1, x1x	WSync	UNPREDICTABLE	-	-
0	0	xxx	Multi-word	Normal	Word[Align(Addr)]	Operation unaffected by Addr[1:0]

Table A2-13 Data Access Behavior in ARMv6 Systems

U	A	Addr[2:0]	Access Types	Behavior	Memory accessed	Notes
0	0	000	Two-word	Normal	Word[Addr]	-
0	0	xx1, x1x, 1xx	Two-word	UNPREDICTABLE	-	-
1	0					NEW ARMv6 UNALIGNED SUPPORT
1	0	xxx	Byte	Normal	Byte[Addr]	-
1	0	xxx	Halfword	Normal	Halfword[Addr]	-
1	0	xxx	WLoad WStore	Normal	Word[Addr]	-
1	0	x00	WSync Multi-word Two-word	Normal	Word[Addr]	-
1	0	xx1, x1x	WSync Multi-word Two-word	Alignment Fault	-	-
x	1					FULL ALIGNMENT FAULTING
x	1	xxx	Byte	Normal	Byte[Addr]	-
x	1	xx0	Halfword	Normal	Halfword[Addr]	-
x	1	xx1	Halfword	Alignment Fault	-	-
x	1	x00	WLoad WStore WSync Multi-word	Normal	Word[Addr]	-
x	1	xx1, x1x	WLoad WStore WSync Multi-word	Alignment Fault	-	-
x	1	000	Two-word	Normal	Word[Addr]	-

Table A2-13 Data Access Behavior in ARMv6 Systems

U	A	Addr[2:0]	Access Types	Behavior	Memory accessed	Notes
0	1	100	Two-word	Alignment Fault	-	-
1	1	100	Two-word	Normal	Word[Addr]	-
x	1	xx1, x1x	Two-word	Alignment Fault	-	-

Other reasons for unaligned accesses to be UNPREDICTABLE

The following exceptions to the behavior described in Table A2-13 on page A2-44 apply, causing the resultant unaligned accesses to be UNPREDICTABLE:

- An LDR instruction that loads the PC, has Addr[1:0] != 0b00, and is specified in the table as having Normal behavior instead has UNPREDICTABLE behavior.

  ——— **Note** ———

  The reason this applies only to LDR is that most other load instructions are UNPREDICTABLE regardless of alignment if the PC is specified as their destination register. The exceptions are LDM, RFE and Thumb POP. If Addr[1:0] != 0b00 for these instructions, the effective address of the transfer has its two least significant bits forced to 0 if A == 0 and U ==0, and otherwise the behavior specified in the table is either UNPREDICTABLE or Alignment Fault regardless of the destination register.

  ————————————

- Any WLoad, WStore, WSync, Two-word or Multi-word instruction that accesses memory with the Strongly Ordered or Device memory attribute, has Addr[1:0] != 0b00, and is specified in the table as having Normal behavior instead has UNPREDICTABLE behavior.

- Any Halfword instruction that accesses memory with the Strongly Ordered or Device memory attribute, has Addr[0] != 0, and is specified in the table as having Normal behavior instead has UNPREDICTABLE behavior.

If any of these reasons applies, it overrides the behavior specified in the table.

——— **Note** ———

These reasons never cause Alignment Fault behavior to be overridden.

————————————

ARM implementations are not required to ensure that the low-order address bits that make an access unaligned are cleared from the address they send to memory. They can instead send the address as calculated by the load/store instruction unchanged to memory, and require the memory system to ignore address[0] for a halfword access and address[1:0] for a word access.

When an instruction ignores the low-order address bits that make an access unaligned, the pseudo-code in the instruction description does not mask them out explicitly. Instead, the Memory[<address>,<size>] function used in the pseudo-code masks them out implicitly. This function is defined in xref TBD

ARMv6 unaligned data access restrictions

ARMv6 has the following restrictions on unaligned data accesses:

- Accesses are not guaranteed atomic. They can be synthesized out of a series of aligned operations in a shared memory system without guaranteeing locked transaction cycles.

- Accesses typically take a number of cycles to complete compared to a naturally aligned transfer. The real-time implications must be carefully analyzed and key data structures might need to have their alignment adjusted for optimum performance.

- Accesses can abort on either or both halves of an access where this occurs over a page boundary. The Data Abort handler must handle restartable aborts carefully after an Alignment Fault Status Code is signalled.

Therefore shared memory schemes should not rely on seeing monotonic updates of non-aligned data of loads, stores, and swaps for data items greater than byte width.

Unaligned access operations should not be used for accessing Device memory-mapped registers. They must also be used with care in shared memory structures that are protected by aligned semaphores or synchronization variables.

A2.9   Synchronization primitives

Historically, support for shared memory synchronization has been with the read-locked-write operations that swap register contents with memory; the SWP and SWPB instructions described in sections *SWP* on page A4-224 and *SWPB* on page A4-226. These support basic busy/free semaphore mechanisms, but not mechanisms that require calculation to be performed on the semaphore between the read and write phases. ARMv6 provides a new mechanism to support more comprehensive non-blocking shared-memory synchronization primitives that scale for multiple-processor system designs.

Two instructions are introduced to the ARM instruction set:

*   Load-Exclusive described in *LDREX* on page A4-51
*   Store-Exclusive described in *STREX* on page A4-208.

The instructions operate in concert with an address monitor, which provides the state machine and associated system control for memory accesses. Two different monitor models exist, depending on whether the memory has the sharable or non-sharable memory attribute. See *Shared attribute* on page B2-13. Uniprocessor systems are only required to support the non-shared model, allowing them to support synchronization primitives with the minimum amount of hardware overhead. An example minimal system monitor is illustrated in Figure A2-15.
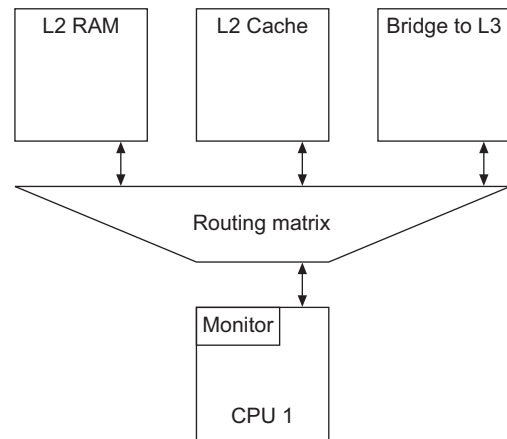
Figure A2-15 Example uniprocessor (non-shared) monitor

Multi-processor systems are required to implement an address monitor for each processor. It is IMPLEMENTATION DEFINED where the monitors reside in the memory system hierarchy, whether they are implemented as a single entity for each processor visible to all shared accesses, or as a distributed entity. Figure Figure A2-16 on page A2-49 illustrates a single entity approach in which the monitor supports state machines for both the shared and non-shared cases. Only the shared attribute case needs to snoop.
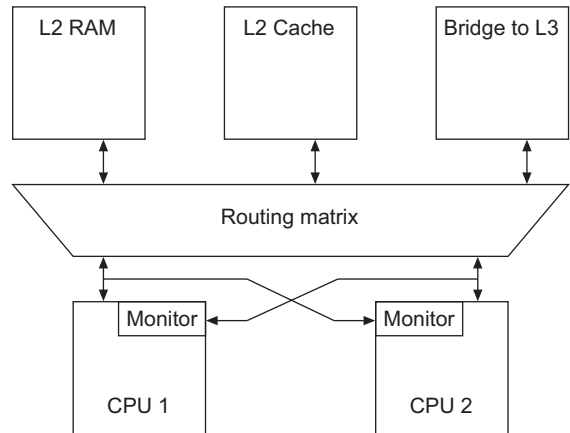
Figure A2-16 Write snoop monitor approach

Figure A2-17 illustrates a distributed model with the non-shared monitors residing in the processor block, and the shared monitors distributed across the targets of interest.
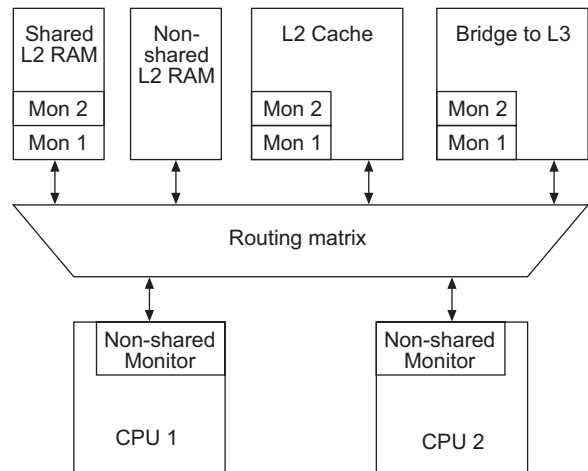
Figure A2-17 Monitor-at-target approach

A2.9.1   Exclusive access instructions: non-shared memory

For memory regions that do not have the *Shared* TLB attribute, the exclusive-access instructions rely on the ability to tag the fact that an exclusive load has been executed. Any non-aborted attempt by the processor that executed the exclusive load to modify any address using an exclusive store is guaranteed to clear this tag.

—— **Note** ——

It is UNPREDICTABLE whether a store to a tagged physical address in non-shared memory by a processor other than the one that caused the physical address to be tagged, will cause the tag to be cleared.

Load-Exclusive performs a load from memory, and causes the executing processor to tag the fact that it has an outstanding tagged physical address to non-sharable memory; the monitor transitions state to Exclusive Access.

Store-Exclusive performs a conditional store to memory, the store only taking place if the executing processor has an outstanding tagged physical address. A status value of 0b0 is returned to a register, and the executing processor's monitor transitions to the Open Access state. If the store is prevented, a value of 0b1 is returned in the instruction defined register.

A write by that processor using any instruction other than a Store-Exclusive will not affect its outstanding tagged physical address.

If a processor performs a Store-Exclusive to any address in non-shared memory other than the last one from which it has performed a Load-Exclusive, the store will succeed, and the executing processor will no longer have any outstanding tagged physical addresses. This mechanism is used on a context switch (see section *Context switch support* on page A2-51). It should be treated as a software programming error in all other cases.

The state machine for the associated data monitor is illustrated in Figure A2-18.
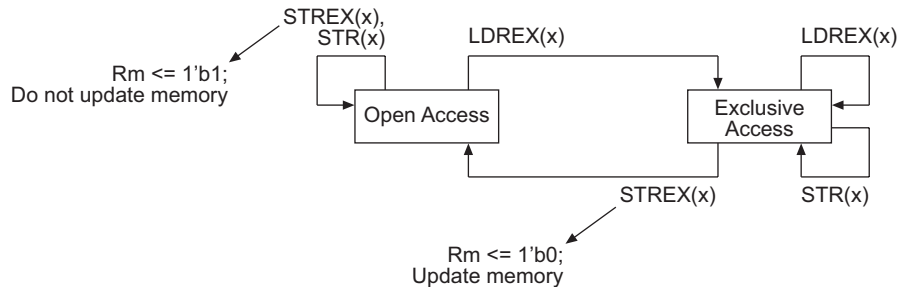
Figure A2-18 FSM noshare

The behavior illustrated is for the address monitor associated with the processor issuing the LDREX, STREX and STR instructions to non-shared memory. It is UNPREDICTABLE whether a STR or STREX memory access from a different processor causes an Exclusive to Open access transition. Transactions from other processors need not be visible to this monitor.

A2.9.2   Exclusive access instructions: shared memory

For memory regions that have the *Shared* TLB attribute, the exclusive-access instructions rely on the ability to tag a physical address as exclusive-access for a particular processor. This tag will later be used to determine whether an exclusive store to that address should occur. Any non-aborted attempt to modify that address by any processor is guaranteed to clear this tag.

Load-Exclusive performs a load from memory, and causes the physical address of the access to be tagged as exclusive-access for the requesting processor. This also causes any other physical address that has been tagged by the requesting processor to no longer be tagged as exclusive access; only a single outstanding exclusive access to sharable memory per processor is supported.

Store-Exclusive performs a conditional store to memory. The store will only take place if the physical address is tagged as exclusive-access for the requesting processor. A status value of 0b0 is returned to a register to acknowledge a successful store, otherwise a value of 0b1 is returned. In both cases, the executing processor's monitor transitions to the Open Access state.

Every processor (or independent DMA agent) in a shared memory system requires its own address monitor. The shared memory state machine for the address monitor associated with a processor (n) in a multiprocessing environment must interact with the memory system in the following ways:

- transactions generated by the associated processor (n)
- transactions associated with other processors in the shared memory system (!n).

This behavior is illustrated in Figure A2-19.
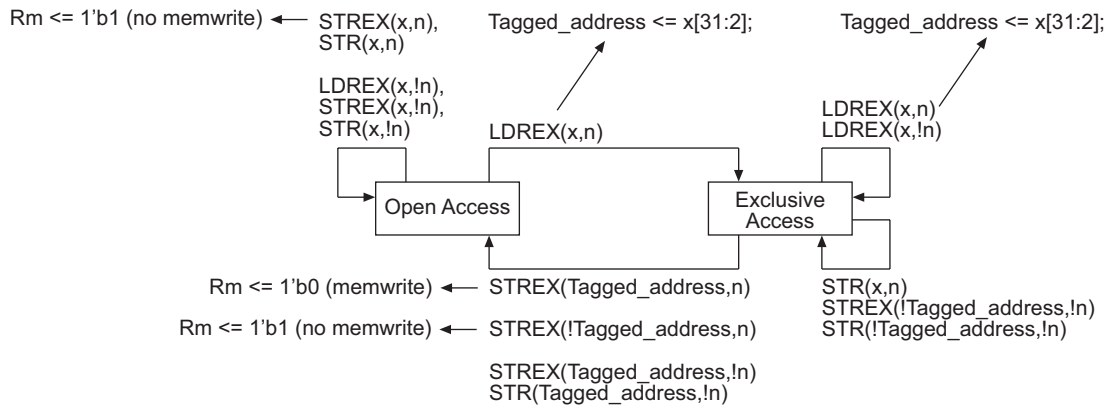
Figure A2-19 FSM shared

――――― **Note** ―――――

Whether a STREX successfully updates memory or not is dependent on a tag address match with its own associated FSM, hence the (!n) entries are only shown with respect to how they influence state transitions of the state machine. Similarly, an LDREX can only update the tag of its own associated FSM.

A2.9.3   Context switch support

On a context switch, it is necessary to ensure all monitors are in the Open Access state after a context switch. This requires execution of a dummy STREX to an address in memory allocated for this purpose. Non-shared memory and shared memory monitors must be cleared down independently, and will require separate

addresses allocated in systems where memory management (an MMU as described in Chapter B4 *Virtual Memory System Architecture* or PMU as described in Chapter B5 *Protected Memory System Architecture*) is enabled.

For reasons of performance, it is recommended that the store-exclusive instruction be within a few instructions of the load-exclusive instruction. This minimizes the opportunity for context switch overhead or multiprocessor access conflicts causing an exclusive store to fail, and requiring the load/store sequence to be replayed.

A2.9.4 Summary of operation

The tables below summaries the operation of load-exclusive, store-exclusive, and store on the physical address tags maintained by the data monitors for uniprocessor and multiprocessor systems. These tables are consistent and could be merged, however, the information has been split out to remove any ambiguity in the uniprocessor state, where the processor ID always assumes a value of zero.

—— **Note** ——

The shared attribute is redundant in a uniprocessor system. It is included to describe how exclusive memory accesses marked as shared shall behave; the processor_id is always zero in the uniprocessor case. Similarly, a null address is stored in the non-shared monitor when in the exclusive access state, and hence marked as zero too.

———————————

Three operations can be performed on the tags:

• Set(physical_address, processor_id), monitor transitions to Exclusive Access state

• Clear(physical_address, processor_id), monitor transitions to Open Access state

• Return(physical_address, processor_id), return success/fail status to target register

Each table entry gives the operations for a processor, P(n), accessing a physical address, A(x).

Table A2-14 Uniprocessor system summary

Operation	A(x) is shared	A(x) is not shared
LDREX from A(x)	Clear(!x,0) Set(x,0)	Set(0,0)
STR to A(x)	No effect	No effect
STREX to A(x)	Return(x,0) Clear(x,0) Clear(!x,0)	Return(0,0) Clear(0,0)

Table A2-15 Multiprocessor system summary

Operation	A(x) is shared	A(x) is not shared
P(n) `LDREX` from A(x)	Clear(!x,n) Set(x,n)	Set(0,n)
P(n) `STR` to A(x)	Clear(x,!n)	UNP whether Clear(0,!n)
P(n) `STREX` to A(x)	Return(x,n) Clear(x,n) Clear(!x,n) IF STREX succeeds, THEN Clear(x,!n) ENDIF	Return(0,n) Clear(0,n) UNP whether Clear(0!n)

The following pseudo-functions can be used to describe the exclusive access operations:

- TLB(<Rn>)
- Shared(<Rn>)
- ExecutingProcessor()
- MarkExclusiveGlobal(<physical_address>,<processor_id>)
- MarkExclusiveLocal(<processor_id>)
- IsExclusiveGlobal(<physical_address>,<processor_id>)
- IsExclusiveLocal(<processor_id>)
- ClearByAddress(<physical_address>)
- ClearOthersByAddress(<physical_address>,<processor_id>)
- ClearByProcessor(<processor_id>)

1. If CP15 register 1 bit[0] (Mbit) is set, TLB(<Rn>) will return the physical address corresponding to the virtual address in Rn for the executing processor's current process ID and TLB entries. If Mbit is not set, it will return the value in Rn. The Mbit describes the control mechanism used to enable an MMU as described in Chapter B4 *Virtual Memory System Architecture*.

2. If CP15 register 1 bit[0] (Mbit) is set, Shared(<Rn>) will return the value of the shared memory region attribute corresponding to the virtual address in Rn for the executing processors current process ID and TLB entries. If Mbit is not set, it will return 0.

3. ExecutingProcessor() returns a value distinct amongst all processors in a given system, corresponding to the processor executing the operation.

4. MarkExclusiveGlobal(<physical_address>,<processor_id>) will record the fact that processor <processor_id> has requested exclusive access to address <physical_address>. This causes any previous request for exclusive access to any other address by the same processor to be cleared.

5. MarkExclusiveLocal(<processor_id>) will record the fact that processor <processor_id> has requested exclusive access to an address.

6. IsExclusiveGlobal(<physical_address>,<processor_id>) will return TRUE if the processor <processor_id> has marked the address <physical_address> as exclusive access requested. Otherwise it will return FALSE.

7.   IsExclusiveLocal(<processor_id>) will return TRUE if the processor <processor_id> has marked an address as exclusive access requested. Otherwise it will return FALSE.

8. ClearByAddress(<physical_address>) will clear any request by any processor to mark address <physical_address> as exclusive access.

9. ClearOthersByAddress(<physical_address>, <processor_id>) will clear any request by any processor other than processor <processor_id> to mark address <physical_address> as exclusive access.

10. ClearByProcessor(<processor_id>) will clear any request by processor <processor_id> to mark any address as exclusive access.

The exclusive accesses can be described in terms of their register file usage:

•   Rd == destination register; data on loads, status on stores

•   Rm == source data register for stores

•   Rn == the memory address register for loads and stores.

and the pseudo-functions outlined as follows.

LDREX operation:

```
    processor_id = ExecutingProcessor()
    Rd = Memory[Rn,4]
    if Shared(Rn) == 1 then
     physical_address = TLB(Rn)
     MarkExclusiveGlobal(physical_address,processor_id)
    else
            MarkExclusiveLocal(processor_id)
    endif
```

STREX operation:

```
    processor_id = ExecutingProcessor()
    if Shared(Rn) == 1 then
        physical_address = TLB(Rn)
        if IsExclusiveGlobal(physical_address,processor_id) then
            Memory[Rn,4] = Rm
            Rd = 0
                ClearByAddress(physical_address)
        else
            Rd = 1
            ClearByProcessor(processor_id)
        endif
    else
        if IsExclusiveLocal(processor_id) then
            Memory[Rn,4] = Rm
```

```
            Rd = 0
        else
            Rd = 1
        endif
            ClearByProcessor(processor_id)
    endif
```

In addition, it should be noted that all instructions with the semantics of a store to memory (see *Writes* on page B2-7) with the Shared TLB attribute now may have a side effect of clearing any tags set by a LDREX instruction, adding the following to the store's semantics for a multiprocessor monitor (they have no effect in the uniprocessor case):

```
if Shared(Rn) == 1 then
        processor_id = ExecutingProcessor()
        physical_address = TLB(Rn)
        ClearOthersByAddress(physical_address,processor_id)
```

Aborted transactions and the Address Monitor

Any load or store transactions which abort shall not affect the state of the shared or non-shared address monitors. Aborted transactions shall be transparent to the synchronizing primitive monitor system, allowing the exception handler to determine the cause of the abort, rectify the problem where it is recoverable, and then replay it.

A2.10   The Java Extension

The Java Extension was first introduced in ARMv5TEJ, a variant of ARMv5, and is a mandated feature in ARMv6. The Java Extension enables architectural support for hardware acceleration of bytecode execution by Java Virtual Machines (JVMs). It is designed in such a way that JVMs can be written to automatically take advantage of any accelerated bytecode execution supplied by the processor, without relying upon it being present. In the simplest implementations, the processor does not accelerate the execution of any bytecodes, and all bytecodes are executed by software routines. This is known as a trivial implementation of the Java Extension, and has minimal costs compared with not implementing the Java Extension at all. Non-trivial implementations of the Java Extension will typically implement a subset of the Java bytecodes in hardware, choosing bytecodes that can have simple hardware implementations and that account for a large percentage of Java execution time.

The required features of a non-trivial implementation are:

*   provision of an additional state bit (the J bit) in the CPSR and each SPSR
*   a new instruction to enter Java state (BXJ)
*   extension of the PC to support full 32-bit byte addressing
*   changes to the exception model
*   mechanisms to allow a JVM to configure the Java Extension hardware to its specific needs
*   mechanisms to allow OSes to regulate use of the Java Extension hardware.

The required features of a trivial implementation are:

*   Only ARM and Thumb execution states shall exist. The J-bit may always read and write as zero. Should the J-bit update to one, execution of the following instruction is UNDEFINED.
*   The BXJ instruction shall behave as a BX instruction.
*   Configuration support that maintains the interface as permanently disabled.

A JVM that has been written to automatically take advantage of hardware-accelerated bytecode execution is known as an enabled JVM (EJVM).

A2.10.1   Subarchitectures

ARM implementations that include the Java Extension expect the ARM processor's general-purpose registers and other resources to obey a calling convention when Java state execution is entered and exited. For example, a specific general-purpose register may be reserved for use as the pointer to the current Java bytecode. In order for an EJVM or associated debug support to function correctly, it must be written to comply with the calling convention expected by the acceleration hardware at Java state execution entry and exit points.

The calling convention is relied upon by an EJVM, but not in general by other system software. This limits the cost of changing the convention to the point that it can be considered worthwhile to change it if a sufficient technical advantage is obtained by doing so, such as a significant performance improvement in Java bytecode execution.

Multiple conventions are known collectively as the subarchitecture of the implementation. They are not described in this document, and must only be relied upon by EJVM implementations and debug/similar software as described above. All other software must only rely upon the general architectural definition of the Java Extension described in this section. A particular subarchitecture is identified by reading the Java ID register described in *Java ID register* on page A2-65.

A2.10.2 Java State

The Java extension makes use of an extra state bit (J) in the processor status registers (the CPSR and the banked SPSRs). This is bit[24] of the registers concerned:

31	30	29	28	27	26 25	24	23 20	19 16	15 10	9	8	7	6	5	4 0
N	Z	C	V	Q	Rsrvd	J	Reserved	GE[3:0]	Reserved	E	A	I	F	T	Mode

The other bit fields are described in section *Program status registers* on page A2-10.

— **Note** —

The placement of the J bit in the flags byte was to avoid any usage of the status or extension bytes in code run on ARMv5TE or earlier processors. This ensures that OS code written using the deprecated CPSR, SPSR, CPSR_all or, SPSR_all syntax for the destination of an MSR instruction only ceases to work when features introduced in ARMv6 are used, namely the E, A and GE bit fields.

In addition, J is always 0 at times that an MSR instruction is executed. This ensures there are no unexpected side-effects of existing instructions such as MSR CPSR_f,#0xF0000000, that are used to put the flags into a known state.

The J bit is used in conjunction with the T bit to determine the execution state of the processor, as shown in Table A2-16.

Table A2-16

J	T	Execution state
0	0	ARM state, executing 32-bit ARM instructions
0	1	Thumb state, executing 16-bit Thumb instructions
1	0	Java state, executing variable-length Java bytecode instructions
1	1	UNDEFINED, and reserved for future expansion

The J bit is treated similarly to the T bit in the following respects:

• On exception entry, both bits are copied from the CPSR to the exception mode's SPSR, and then cleared in the CPSR to put the processor into the ARM state.

- Data processing instructions with Rd = R15 and the S bit set cause these bits to be copied from the SPSR to the CPSR and execution to resume in the resulting state. This ensures that these instructions have their normal exception return functionality.

  Such exception returns are expected to use the SPSR and R14 values generated by a processor exception entry and to use the appropriate return instruction for the exception concerned, as described in *Exceptions* on page A2-15. If return values are used with J == 1 and T == 0 in the SPSR value, then the results are SUBARCHITECTURE DEFINED.

- Similarly, LDM instructions with the PC in the register list and ^ specified (that is, LDM (3) instructions, as described in section *LDM (3)* on page A4-39) cause both bits to be copied from the SPSR to the CPSR and execution to resume in the resulting state. These instructions are also used for exception returns, and the considerations in the previous bullet point also apply to them.

- In privileged modes, execution of an MSR instruction that attempts to set the J or T bit of the CPSR to 1 has UNPREDICTABLE results.

- In unprivileged (User) mode, execution of an MSR instruction that attempts to set the J or T bit of the CPSR to 1 will not modify the bit.

- Setting J == 1 and T == 1 causes similar effects to setting T == 1 on a non Thumb-aware processor. That is, the next instruction executed will cause entry to the Undefined Instruction exception. Entry to the exception handler will cause the processor to re-enter ARM state, and the handler can detect that this was the cause of the exception because J and T are both set in SPSR_und.

While in Java state, the processor executes Java bytecode programs. A Java bytecode program is defined to be an executable object comprising one or more Java class files, as defined in Lindholm and Yellin, *The Java Virtual Machine Specification 2nd Edition*, or derived from and functionally equivalent to one or more Java class files. While in Java state, the PC acts as a program counter which identifies the next Java bytecode instruction to be executed, where Java bytecode instructions are the bytecode instructions defined in Lindholm and Yellin, or a functionally equivalent transformed version of them.

Native methods, as described in Lindholm and Yellin, for the Java Extension must use only the ARM and/or Thumb instruction sets to specify their functionality.

An implementation of the Java Extension must not be documented or promoted as performing any task while it is in Java state other than the acceleration of Java bytecode programs in accordance with this section and Lindholm and Yellin.

Extension of the PC to 32 bits

In order to allow the PC to point to an arbitrary Java bytecode, all 32 bits of the PC are defined in non-trivial implementations. Bit[0] of the PC always reads as zero when in ARM or Thumb state. Bit[1] reflects the word-alignment, or halfword-alignment of ARM and Thumb instructions respectively. The existence of bit[0] in the PC is only visible in ARM or Thumb state due to an exception occurring in Java state, and the exception return address is odd-byte aligned.

The main architectural implication of this is that exception handlers must ensure that they restore all 32 bits of R15. The recommended ways to handle exception returns behave correctly.

A2.10.3   New Java state entry instruction (BXJ)

An ARM instruction similar to BX is added. The BXJ instruction has a single register operand that specifies a target execution state (ARM or Thumb) and branch target address for use if entry to Java state is not available. See *BXJ* on page A4-20 for more details.

Compliant Java execution involves the EJVM using the BXJ instruction, the usage model of the standard ARM registers, and the Java Extension Control and Configuration registers described in *Configuration and control* on page A2-64.

Executing BXJ with Java extension enabled

Executing a BXJ instruction when the JE bit is 1 gives the Java Extension hardware an opportunity to enter Java state and start executing Java bytecodes directly. The circumstances in which Java state execution is entered are IMPLEMENTATION DEFINED. If Java state execution is not entered, the instruction is executed in the same way as a BX instruction to a SUBARCHITECTURE DEFINED register usage model. This is required to ensure the Java Extension hardware and the EJVM software communicate effectively with each other. Similarly, various registers will contain SUBARCHITECTURE DEFINED values when Java state execution is terminated and ARM or Thumb state execution is resumed. The precise set of registers affected by these requirements is a SUBARCHITECTURE DEFINED subset of the process registers, which are defined to be:

- the ARM general-purpose registers R0-R14
- the PC
- the CPSR
- the VFP general-purpose registers S0-S31 and D0-D15, subject to the VFP architecture's restrictions on their use and subject to the VFP architecture being present
- the FPSCR, subject to the VFP architecture being present.

All processor state that can be modified by Java state execution must be kept in process registers, in order to ensure that it is preserved and restored correctly when processor exceptions and process swaps occur. Configuration state (that is, state that affects Java state execution but is not modified by it) can be kept either in process registers or in configuration registers.

EJVM implementations should only set JE == 1 after determining that the processor's Java Extension subarchitecture is compatible with their usage of the process registers. Otherwise, they should leave JE == 0 and execute without hardware acceleration.

Executing BXJ with Java extension disabled

If a BXJ instruction is executed when the JE bit is 0, it is executed identically to a BX instruction with the same register operand.

BXJ instructions can therefore be freely executed when the JE bit is 0. In particular, if an EJVM determines that it is executing on a processor whose Java Extension implementation is trivial or uses an incompatible subarchitecture, it can set JE == 0 and execute correctly, without the benefit of any bytecode hardware acceleration that may be present.

Java state exit

The processor exits Java state in IMPLEMENTATION DEFINED circumstances. This is typically due to attempted execution of a bytecode that the implementation cannot handle in hardware, or that generates a Java exception (such as a NullPointerException). When this occurs, various processor registers will contain SUBARCHITECTURE DEFINED values, allowing the EJVM to resume software execution of the bytecode program correctly.

The processor also exits Java state when a processor exception occurs. The CPSR is copied to the exception mode's banked SPSR as normal, so the banked SPSR contains J == 1 and T == 0, and Java state is restored on return from the exception when the SPSR is copied back into the CPSR. Coupled with the restriction that only process registers can be modified by Java state execution, this ensures that all registers are correctly preserved and restored by processor exception handlers. Configuration and control registers may be modified in the exception handler itself as described in section *Configuration and control* on page A2-64.

Considerations specific to execution of Java bytecodes apply to processor exceptions. For details of these, see *Java Extension exception handling* on page A2-61.

It is IMPLEMENTATION DEFINED whether Java Extension hardware contains state that is modified during Java state execution, and is held outside the process registers during Java state execution. If such state exists, the implementation shall:

•   Initialize the state from one or more of the process registers whenever Java state is entered, either as a result of execution of a BXJ instruction or of returning from a processor exception.

•   Write the state into one or more of the process registers whenever Java state is exited, either as a result of taking a processor exception or of IMPLEMENTATION DEFINED circumstances.

•   Ensure that the ways in which it is written into process registers on taking a processor exception, and initialized from process registers on returning from that exception, result in it being correctly preserved and restored over the exception.

Additional Java state restrictions

The Java Extension hardware shall obey the following restrictions:

•   It must not change processor mode other than by taking one of the standard ARM processor exceptions.

•   It must not access banked versions of registers other than the ones belonging to the processor mode in which it is entered.

•   It must not do anything that is illegal for an UNPREDICTABLE instruction. That is, it must not generate a security loophole, nor halt or hang the processor or any other part of the system.

As a result of these requirements, Java state can be entered from User mode without risking a breach of OS security. In addition:

•   Entering Java state from FIQ mode has UNPREDICTABLE results.

- Java Extension subarchitectures and implementations must not make use of otherwise-unallocated CPSR and SPSR bits. All such bits are reserved for future expansion of the ARM and Thumb architectures.

A2.10.4 Java Extension exception handling

All exceptions copy the J bit from the CPSR to the SPSR, and all instructions that have the side-effect of copying the SPSR to the CPSR must copy the J bit along with all the other bits.

When an exception occurs in Java state, the R14 register for the exception mode is calculated as follows:

IRQ/FIQ   Address of bytecode to be executed on return from interrupt + 4.

Prefetch Abort Address of the bytecode causing the abort + 4.

Data Abort   Address of the bytecode causing the abort + 8.

Undefined instruction

Must not occur. See *Undefined instruction exceptions* on page A2-63.

SWI   Must not occur. See *SWI exceptions* on page A2-63.

Interrupts (IRQ and FIQ)

In order for the standard mechanism for handling interrupts to work correctly, Java Exception hardware implementations must take care that whenever an interrupt is allowed to occur during Java state execution, one of the following occurs:

- Execution has reached a bytecode instruction boundary. That is, all operations required to implement one bytecode instruction have completed, and none of the operations required to implement the next bytecode instruction have completed. The R14 value on entry to the interrupt handler must be the address of the next bytecode instruction, plus 4.

- The sequence of operations performed from the start of the current bytecode instruction's execution up to any point where an interrupt can occur is idempotent: that is, it can be repeated from its start without changing the overall result of executing the bytecode. The R14 value on entry to the interrupt handler must be the address of the current bytecode instruction, plus 4.

- If an interrupt does occur during a bytecode's execution, corrective action is taken either directly by the Java Extension hardware or indirectly by it calling a SUBARCHITECTURE DEFINED handler in the EJVM, and that corrective action re-creates a situation in which the bytecode instruction can be re-executed from its start. The R14 value on entry to the interrupt handler must be the address of the bytecode instruction, plus 4.

Data aborts

The value saved in R14_abt on a data abort shall ensure that a virtual memory data abort handler can read the system coprocessor (CP15) Fault Status and Fault Address Registers, fix the reason for the abort and return using `SUBS PC,R14,#8` or its equivalent, without looking at the instruction that caused the abort or which state it was executed in.

——— **Note** ———

This assumes that the intention is to return to and retry the bytecode instruction that caused the data abort. If the intention is instead to return to the bytecode instruction after the one that caused the abort, then the return address will need to be modified by the length of the bytecode instruction that caused the abort.

In order for the standard mechanism for handling data aborts to work correctly, Java Exception hardware implementations must ensure that one of the following applies where bytecode instruction might generate a data abort:

- The sequence of operations performed from the start of the bytecode instruction's execution up to the point where the data abort occurs is idempotent. That is, it can be repeated from its start without changing the overall result of executing the bytecode instruction.

- If the data abort occurs during bytecode execution, corrective action is taken either directly by the Java Extension hardware or indirectly by it calling a SUBARCHITECTURE DEFINED handler in the EJVM, and that corrective action re-creates a situation in which the bytecode instruction can be re-executed from its start.

——— **Note** ———

In ARMv6, the Base Updated Abort Model is no longer allowed (see *Abort models* on page A2-22). This removes one potential obstacle to the first of these solutions.

Prefetch aborts

The value saved in R14_abt on a prefetch abort shall ensure that a virtual memory prefetch abort handler can locate the start of the instruction that caused the abort simply and without looking at the state in which its execution was attempted. It is always at address (R14_abt – 4).

However, a multi-byte Java bytecode instruction may cross a page boundary, in which case the ARM processor's prefetch abort handler cannot determine directly which of the two pages caused the abort. It is SUBARCHITECTURE DEFINED how this situation is handled, subject to the requirement that if it is handled by calling the ARM processor's prefetch abort handler, (R14_abt – 4) must point to the first byte of the bytecode instruction concerned.

In order to ensure subarchitecture-independence, OS designers should write prefetch abort handlers in such a way that they can handle a prefetch abort generated in either of the two pages spanned by such a bytecode instruction. A suggested simple technique is:

```
IF the page pointed to by (R14_abt - 4) is not mapped
    THEN map the page
    ELSE map the page following the page including (R14_abt - 4)
ENDIF
retry the instruction
```

SWI exceptions

SWI exceptions must not occur during Java state execution, for the following reasons:

- ARM and Thumb state SWIs are supported in the ARM architecture, bytecode SWIs are not due to the additional complexity in the SWI usage model they would introduce.

- Java extension subarchitectures and implementations need to have a mechanism to return to ARM or Thumb state handlers in order to execute the more complex bytecode instructions. If a bytecode instruction needs to make an OS call, it can make use of this mechanism to cause an ARM or Thumb SWI instruction to be executed, with a small overhead in percentage terms compared with the cost of the OS call itself.

- SWI calling conventions are highly OS-dependent, and would potentially require the subarchitecture to be OS aware.

Undefined instruction exceptions

Undefined instruction exceptions must not occur during Java state execution.

When the Java Extension hardware synthesizes a coprocessor instruction and passes it to a hardware coprocessor (most likely, a VFP coprocessor), and the coprocessor rejects the instruction, there are considerable complications involved if this was allowed to result in the ARM processor's undefined instruction trap. These include:

- The coprocessor instruction is not available to be loaded from memory (something that is relied upon by most undefined instruction handlers).

- The coprocessor instruction cannot typically be determined from the bytecode instruction that is loadable from memory without considerable knowledge of implementation and subarchitecture details of the Java Extension hardware.

- The coprocessor-generated undefined instruction exceptions (and VFP-generated ones in particular) can typically be either precise (that is, caused by the instruction at (R14_und – 4)) or imprecise (that is, caused by a pending exceptional condition generated by some earlier instruction and nothing to do with the instruction at (R14_und – 4)).

  Precise undefined instruction exceptions typically need to be handled by emulating the instruction at (R14_und – 4), followed by returning to the instruction that follows it. Imprecise undefined instruction exceptions typically need to be handled by getting details of the exceptional condition and/or the earlier instruction from the coprocessor, fixing things up in some way, and then returning to the instruction at (R14_und – 4).

This means that there are two different possible return addresses, not necessarily at a fixed offset from each other as they are when dealing with coprocessor instructions in memory, making it difficult to define the value R14_und should have on entry to the undefined instruction handler.

- The return address for the undefined instruction handler places idempotency requirements and/or completion requirements (that is, that once the coprocessor operation has been completed, everything necessary for execution of the bytecode instruction has been done) on the sequences of operations performed by the Java Extension hardware.

  The restrictions require cooperation and limit the design freedom for both the Java acceleration and coprocessor designers.

To avoid the need for undefined exceptions, the following coprocessor interworking model for Java Extension hardware applies.

Coprocessor Interworking

If while executing in Java state, the Java Extension hardware synthesizes a coprocessor instruction and passes it to a hardware coprocessor for execution, then it must be prepared for the coprocessor to reject the instruction. If a coprocessor rejects an instruction issued by Java Extension hardware, the Java Extension hardware and coprocessor must cooperate to:

- Prevent the undefined instruction exception that would occur if the coprocessor had rejected a coprocessor instruction in ARM state from occurring.

- Take suitable SUBARCHITECTURE DEFINED corrective action, probably involving exiting Java state, and executing a suitable ARM code handler that contains further coprocessor instructions.

To ensure that this is a practical technique and does not result in inadequate or excessive handling of coprocessor instruction rejections, coprocessors designed for use with the Java Extension must:

- When there is an exceptional condition generated by an earlier instruction, the coprocessor shall keep track of that exceptional condition and keep trying to cause an imprecise undefined instruction exception whenever an attempt is made to execute one of its coprocessor instructions until the exceptional condition is cleared by its undefined instruction handler.

- When it tries to cause a precise undefined instruction exception, for reasons to do with the coprocessor instruction it is currently being asked to execute, the coprocessor shall act in a memoryless way. That is, if it is subsequently asked to execute a different coprocessor instruction, it must ignore the instruction it first tried to reject precisely and instead determine whether the new instruction needs to be rejected precisely.

A2.10.5 Configuration and control

All registers associated with the Java extension are implemented in coprocessor space as part of coprocessor fourteen (CP14). The registers are accessed using the MCR (*MCR* on page A4-61) and MRC (*MRC* on page A4-69) instructions.

The general instruction formats for Java Extension control and configuration are as follows:

```
MCR{<cond>} p14, 7, <Rd>, CRn, CRm{,  opcode_2}*
MRC{<cond>} p14, 7, <Rd>, CRn, CRm{,  opcode_2}*
```

*opcode_2 can be omitted if opcode_2 == 0

The following rules apply to the Java Extension control and configuration registers:

*   All SUBARCHITECTURE DEFINED configuration registers are accessed by coprocessor 14 MRC and MCR instructions with <opcode_1> set to 7.

*   The values contained by configuration registers are only changed by the execution of MCR instructions, and in particular are not changed by Java state execution of bytecodes.

*   The access policy for the required registers is fully defined in their descriptions. All MCR accesses to the Java ID register, and MRC or MCR accesses which are restricted to privileged modes only are UNDEFINED if executed in User mode.

    The access policy of other configuration registers is SUBARCHITECTURE DEFINED.

*   When a configuration register is readable, the result of reading it will be the last value written to it, with no side-effects. When a configuration register is not readable, the result of attempting to read it is UNPREDICTABLE.

*   When a configuration register can be written, the effect must be idempotent. That is, the overall effect of writing the value more than once must not differ from the effect of writing it once.

A minimum of three registers are required in a non-trivial implementation. Additional registers may be provided and are SUBARCHITECTURE DEFINED.

Java ID register

The Java Identity register allows EJVMs to determine the architecture and subarchitecture under which they are running. This is a coprocessor 14 read-only register, accessed by the MRC instruction:

```
MRC{<cond>} p14, 7, <Rd>, c0, c0 {, 0}   ;<Rd>:= Java Identity register
```

The Java ID register is normally accessible from both privileged and User modes. See *Operating System (OS) control register* on page A2-66 for User mode access restrictions.

The format of the Java Identity register is:

31 28	27 20	19 12	11 0
Architecture	Implementor	Subarchitecture	SUBARCHITECTURE DEFINED

Bits[31:28]   Contain an architecture code. This uses the same architecture code that appears in the Main ID register in coprocessor 15

Bits[27:20]   Contain the implementor code of the designer of the subarchitecture. This uses the same implementor code that appears in the Main ID register in coprocessor 15, as documented in section *Main ID register* on page B3-7.

As a special case, if the trivial implementation of the Java Extension is used, this implementor code is 0x00.

Bits[19:12]   Contain the subarchitecture code. The following subarchitecture code is defined:

0x00 = Jazelle V1 subarchitecture, or trivial implementation of Java Extension if implementor code is 0x00.

Bits[11:0]   Contain further SUBARCHITECTURE DEFINED information.

Main configuration register

A Main Configuration register is added to control the Java Extension. This is a coprocessor 14 register, accessed by MRC and MCR instructions as follows:

```
MRC{<cond>}  p14, 7, <Rd>, c2, c0 {, 0}    ; <Rd> := Main Configuration
                                           ; register
MCR{<cond>}  p14, 7, <Rd>, c2, c0 {, 0}    ; Main Configuration
                                           ; register := <Rd>
```

This register is normally write-only from User mode. See *Operating System (OS) control register* for additional User mode access restrictions.

The format of the Main Configuration register is:

31	1	0
SUBARCHITECTURE DEFINED		JE

Bit[31:1]   SUBARCHITECTURE DEFINED information.

Bit[0]   The Java Enable (JE) bit, which is cleared to 0 on reset.

When the JE bit is 0, the Java Extension is disabled and the BXJ instruction does not cause Java state execution – instead, BXJ behaves exactly as a BX instruction. See *BXJ* on page A4-20.

When the JE bit is 1, the Java Extension is enabled.

Operating System (OS) control register

The Java OS Control register provides the operating system with process usage control of the Java Extension. This is a coprocessor 14 register, accessed by MRC and MCR instructions as follows:

```
MRC{<cond>}  p14, 7, <Rd>, c1, c0 {, 0}    ; <Rd> := Java OS Control
                                           ; register
MCR{<cond>}  p14, 7, <Rd>, c1, c0 {, 0}    ; Java OS Control
                                           ; register := <Rd>
```

This register can only be accessed from privileged modes; these instructions are UNDEFINED when executed in User mode. EJVMs will normally never access the Java OS Control register, and EJVMs that are intended to run in User mode cannot do so.

The purpose of the Java OS Control register is primarily to allow operating systems to control access to the Java Extension hardware in a subarchitecture-independent fashion. It is expected to be used in conjunction with the JE bit of the Main Configuration register.

The format of the Java OS Control register is:

31		2	1	0
	Reserved (RAZ)		C V	C D
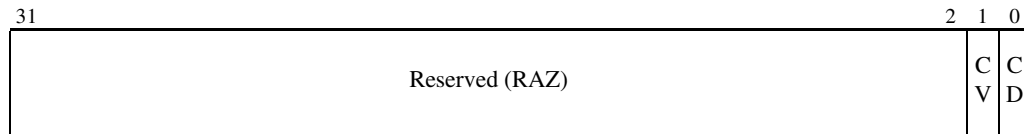
Bits[31:2]   Reserved for future expansion. Prior to such expansion, they must read as zero. To maximize future compatibility, software should preserve their contents, using a read modify write method to update the other control bits.

CV Bit[1]   The Configuration Valid bit, which can be used by an operating system to signal to an EJVM that it needs to re-write its configuration to the configuration registers. When CV == 0, re-writing of the configuration registers is required before a bytecode is next executed. When CV == 1, no re-writing of the configuration registers is required, other than re-writing that is certain to occur before a bytecode is next executed.

CD Bit[0]   The Configuration Disabled bit, which can be used by an operating system to monitor and/or control User mode access to the configuration registers and the Java Identity register. When CD == 0, MCR instructions that write to configuration registers and MRC instructions that read the Java Identity register execute normally. When CD == 1, all of these instructions only behave normally when executed in a privileged mode, and are UNDEFINED when executed in User mode.

When the JE bit of the Main Configuration register is 0, the Java OS Control register has no effect on how BXJ instructions are executed. They always execute as a BX instruction.

When the JE bit of the Main Configuration register is 1, the CV bit affects BXJ instructions as follows:

*   If CV == 1, the Java Extension hardware configuration is considered enabled and valid, allowing the processor to enter Java state and execute bytecodes as described in *Executing BXJ with Java extension enabled* on page A2-59.

*   If CV == 0, then in all of the IMPLEMENTATION DEFINED circumstances in which the Java Extension hardware would have entered Java state if CV had been 1, it instead enters a configuration invalid handler and sets CV to 1. A configuration invalid handler is a sequence of ARM instructions that includes MCR instructions to write the configuration required by the EJVM, ending with a BXJ instruction to re-attempt execution of the bytecode concerned. The method by which the configuration invalid handler's address is determined and its entry and exit conditions are all SUBARCHITECTURE DEFINED.

    In circumstances in which the Java Extension hardware would not have entered Java state if CV had been 1, it is IMPLEMENTATION DEFINED whether the configuration invalid handler is entered as described in the last paragraph, or the BXJ instruction is treated as a BX instruction with possible SUBARCHITECTURE DEFINED restrictions.

The intended use of the CV bit is that when a process swap occurs, the operating system sets CV to 0. The result is that before the new process can execute a Java bytecode in the Java Extension hardware, it must execute its configuration invalid handler. This ensures that the Java Extension hardware's configuration registers are correctly for the EJVM concerned. The CV bit is set to 1 on entry to the configuration invalid handler, allowing the bytecode to be executed in hardware when the invalid configuration handler re-attempts its execution.

——— **Note** ———

It may seem counterintuitive that the CV bit is set to 1 on entry to the configuration invalid handler, rather than after it has completed writing the configuration registers. This is correct, otherwise, the configuration invalid handler may partially configure the hardware before a process swap occurs, causing another EJVM-using process to write its configuration to the hardware.

When the original process is resumed, CV will have been cleared (CV == 0) by the operating system. If the handler writes its configuration to the hardware and then sets CV to 1 in this example, the bytecode will be executed with the hardware configured for a hybrid of the two configurations.

By setting CV to 1 on entry to the configuration invalid handler, this means that CV is 0 when execution of the bytecode is re-attempted, and the configuration invalid handler will execute again (and if necessary, recursively) until it finally completes execution without a process swap occurring.

The CD bit has multiple possible uses for monitoring and controlling User mode access to the Java Extension hardware. Among them are:

*   By setting CD == 1 and JE == 0, an OS can prevent all User mode access to the Java Extension hardware: any attempt to use the BXJ instruction will produce the same result as a BX instruction, and any attempt to configure the hardware (including setting the JE bit) will result in an undefined instruction exception.

*   To provide User mode access to the Java Extension hardware in a simple manner, while protecting EJVMs from conflicting use of the hardware by other processes, the OS should set CD == 0 and should preserve and restore the Main Configuration register on process swaps, initializing its value to 0 for new processes. In addition, it should set the CV bit to 0 on every process swap, to ensure that EJVMs reconfigure the Java Extension hardware to match their requirements when necessary.

*   The technique described in the previous bullet point may result in large numbers of unnecessary reconfigurations of the Java Extension hardware if only a few processes are using the hardware. This can be improved by the OS keeping track of which User mode processes are known to be using an EJVM.

    The OS should set CD == 1 and JE == 0 for any new processes or on a context switch to an existing process that is not using an EJVM. Any User mode instruction that attempts to access a configuration register will take an UNDEFINED exception. The undefined instruction handler can then identify the EJVM need, mark the process as using an EJVM, then return to retry the instruction with CD == 0.

    A further refinement is to clear the CV bit to 0 only if the context switch is to an EJVM-using process that is different from the last EVJM-using process which ran. This avoids redundant reconfiguration of the hardware. That is, the operating system maintains a "process currently owning the Java

Extension hardware" variable, that gets updated with a process_ID when swapping to an EJVM-using process. The context switch software sets CV to 0 if the process_ID update results in a change to the saved variable.

Context switch software implementing the CV-bit scheme should also save and restore the Main Configuration register (in its entirety) on a process swap where the EJVM-using process changes. This ensures that the restored EJVM can use the JE bit reliably for its own purpose.

———— **Note** ————

This technique will not identify privileged EJVM-using processes. However, it is assumed that operating systems are aware of the needs of their privileged processes.

• The OS can impose a single Java Extension configuration on all User mode code by writing that configuration to the hardware, then setting CD == 1 and JE == 1.

The CV and CD bits are both set to 0 on reset. This ensures that subject to some conditions, an EJVM can operate correctly under an OS that does not support the Java Extension. The main such condition is that a process swap never swaps between two EJVM-using processes that require different settings of the configuration registers. This would occur in either of the following two cases, for example:

• if there is only ever one EJVM-using process in the system.
• if all of the EJVM-using processes in the system use the same static settings of the configuration registers.

A2.10.6 EJVM operation

This section summarizes how EJVMs should operate in order to meet the architecture requirements.

Initialization

During initialization, the EJVM should first check which subarchitecture is present, using the implementor and subarchitecture codes in the value read from the Java Identity register.

If the EJVM is incompatible with the subarchitecture, it should either write a value with JE == 0 to the Main Configuration register, or (if unaccelerated bytecode execution is unacceptable) generate an error.

If the EJVM is compatible with the subarchitecture, it should write its desired configuration to the Main Configuration register and any other configuration registers. The EJVM should not skip this step on the assumption that the CV bit of the Java OS Control register will be 0; an assumption that CV == 0 triggering the configuration invalid handler before any bytecode is executed by the Java extension hardware should not be relied on.

Bytecode execution

The EJVM should contain a handler for each bytecode and for each exception condition specified by the subarchitecture it is designed for (the exception conditions always include configuration invalid). It should initiate bytecode execution by executing a BXJ instruction with the register operand specifying the target address of the bytecode handler for the first bytecode of the program, and the process registers set up in accordance with the SUBARCHITECTURE DEFINED register usage model.

The bytecode handler performs the data-processing operations required by the bytecode concerned, determines the address of the next bytecode to be executed, determines the address of the handler for that bytecode, and performs a BXJ to that handler address with the registers again set up to the SUBARCHITECTURE DEFINED register usage model.

The register usage model on entry to exception condition handlers are SUBARCHITECTURE DEFINED, and may differ from the register usage model defined for BXJ instruction execution. The handlers then resolve the exception condition. For example, in the case of the configuration invalid handler, the handler rewrites the desired configuration to the Main Configuration register and any other configuration registers).

Further considerations

To ensure application execution and correct interaction with an operating system, EJVMs should only perform operations that are allowed in User mode. In particular, they should only ever read the Java ID register, write to the configuration registers, and should not attempt to access the Java OS Control register.

A2.10.7 Trivial implementations

This section summarizes what needs to be implemented in trivial implementations of the Java Extension.

- Implement the Java Identity register with the implementor and subarchitecture fields set to zero; the whole register may RAZ (read as zero).

- Implement the Main Configuration register to read as zero and ignore writes.

- Implement the Java OS control register such that it can be read and written, but its effects are ignored. The register may be implemented as RAZ/DNM - read as zero, do not modify on writes. This allows operating systems supporting an EJVM to execute correctly.

- Implement the BXJ instruction to behave identically to the BX instruction in all circumstances, as implied by the fact that the JE bit is always zero. In particular, this means that Java state will never be entered normally on a trivial implementation.

- A trivial implementation may implement the J-bit in the CPSR/SPSRs as RAZ/DNM; read as zero, do not modify on writes. This is allowed because there is no legitimate way to set the J-bit and enter Java state, hence any return routine that tries to do so is issuing an UNPREDICTABLE instruction.

   Otherwise, implement J bits in the CPSR and each SPSR, and ensure that they are read, written and copied correctly when exceptions are entered and when MSR, MRS and exception return instructions are executed.

- In all cases when J == 1 in the CPSR it is IMPLEMENTATION DEFINED whether the next instruction is UNDEFINED or issues a prefetch abort (a similar effect to that caused by setting T == 1 on a non Thumb-aware processor).

———— **Note** ————

The PC does not need to be extended to 32 bits in the trivial implementation, since the only way that bit[0] of the PC is visible in ARM or Thumb state is as a result of a processor exception occurring during Java state execution, and Java state execution does not occur on a trivial implementation.

Chapter A3
The ARM Instruction Set

This chapter describes the ARM instruction set and contains the following sections:

A3.1 Instruction set encoding

Figure A3-1 shows the ARM instruction set encoding.

All other bit patterns are UNPREDICTABLE or UNDEFINED. See *Extending the instruction set* on page A3-35 for a description of the cases where instructions are UNDEFINED.

An entry in square brackets, for example [1], indicates that more information is given after the figure.

	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
Data processing immediate shift	cond [1]	0 0 0	opcode	S	Rn	Rd	shift amount	shift	0	Rm
Miscellaneous instructions: See Figure 3-4	cond [1]	0 0 0	1 0 x x	0	x x x x	x x x x	x x x x	0	x x	x x x x
Data processing register shift [2]	cond [1]	0 0 0	opcode	S	Rn	Rd	Rs	0	shift	1	Rm
Miscellaneous instructions: See Figure 3-4	cond [1]	0 0 0	1 0 x x	0	x x x x	x x x x	x x x x	0	x x	1	x x x x
Multiplies, extra load/stores: See Figure 3-2	cond [1]	0 0 0	x x x x x	x x x x x x x x x x x x	1	x x	1	x x x x
Data processing immediate [2]	cond [1]	0 0 1	opcode	S	Rn	Rd	rotate	immediate
Undefined instruction [3]	cond [1]	0 0 1	1 0 x 0 0	x x x x x x x x x x x x x x x x x x x x
Move immediate to status register	cond [1]	0 0 1	1 0 R 1 0	Mask	SBO	rotate	immediate
Load/store immediate offset	cond [1]	0 1 0	P U B W L	Rn	Rd	immediate
Load/store register offset	cond [1]	0 1 1	P U B W L	Rn	Rd	shift amount	shift	0	Rm
Miscellaneous instructions: See Figure 3-5	cond [1]	0 1 1	x x x x x x x x x x x x x x x x x x x x	1	x x x x
Load/store multiple	cond [1]	1 0 0	P U S W L	Rn	register list
Branch and branch with link	cond [1]	1 0 1 L	24-bit offset
Coprocessor load/store and double register transfers	cond [4]	1 1 0	P U N W L	Rn	CRd	cp_num	8-bit offset
Coprocessor data processing	cond [4]	1 1 1 0	opcode1	CRn	CRd	cp_num	opcode2	0	CRm
Coprocessor register transfers	cond [4]	1 1 1 0	opcode1	L	CRn	Rd	cp_num	opcode2	1	CRm
Software interrupt	cond [1]	1 1 1 1	swi number
Unconditional instructions: See Figure 3-2	1 1 1 1	x x x x x x x x x x x x x x x x x x x x x x x x x x x x

Figure A3-1 ARM instruction set summary

1. The cond field is not allowed to be 1111 in this line. Other lines deal with the cases where bits[31:28] of the instruction are 1111.
2. If the opcode field is of the form 10xx and the S field is 0, one of the following lines applies instead.
3. UNPREDICTABLE prior to ARMv4.
4. If the cond field is 1111, this instruction is UNPREDICTABLE prior to ARMv5.

ARM DDI 0100F-02

A3.1.1 Unconditional instructions

Figure A3-2 shows unconditional instructions. An entry in square brackets, for example [1], indicates that more information is given below the figure.

	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
Change Processor State	1 1 1 1	0 0 0 1 0 0 0 0	imod	M	0	SBZ	A	I	F	0	mode
Set Endianness	1 1 1 1	0 0 0 1 0 0 0 0	0 0 0 1	SBZ	E	SBZ	0 0 0 0	SBZ		
Cache Preload	1 1 1 1	0 1 X 1 U 1 0 1	Rn	1 1 1 1	addr_mode [3]					
Save Return State	1 1 1 1	1 0 0 P U 1 W 0	1 1 0 1	SBZ	0 1 0 1	SBZ	mode			
Return From Exception	1 1 1 1	1 0 0 P U 0 W 0	Rn	SBZ	1 0 1 0	SBZ				
Branch and Branch with Link and change to Thumb [1]	1 1 1 1	1 0 1 H	24-bit offset							
Additional coprocessor double register transfer [1]	1 1 1 1	1 1 0 0 0 0 1 0	L	Rn	Rd	cp_num	opcode	CRm			
Additional coprocessor register transfer [1]	1 1 1 1	1 1 1 0	opc1	L	CRn	Rd	cp_num	opc2	1	CRm
Undefined instruction [2]	1 1 1 1	1 1 1 1	x x x x x x x x x x x x x x x x x x x x x x x x							

Figure A3-2 Unconditional instructions

1. UNDEFINED in architecture 5, UNPREDICTABLE prior to ARMv5.
2. UNPREDICTABLE prior to ARMv5.
3. If X=1 and bit[4]=1, the instruction is UNDEFINED.

M   mmod

X   In addressing mode 2, X=0 implies an immediate offset/index, and X=1 a register based offset/index.

——— **Note** ———

Any instruction with bits[31:28] = 1111, which is not specified in Figure A3-2 or its notes, is an UNDEFINED instruction (or UNPREDICTABLE prior to ARMv4).

A3.1.2 Multiplies and extra load/store instructions

Figure A3-3 on page A3-4 shows multiplies and extra load/store instructions.

An entry in square brackets, for example [1], indicates that more information is given below the figure.

	31 30 29 28	27	26	25	24	23	22	21	20	19–16	15–12	11–8	7	6	5	4	3 2 1 0
Multiply (acc)	cond	0	0	0	0	0	0	A	S	Rd	Rn	Rs	1	0	0	1	Rm
Unsigned multiply acc acc long [3]	cond	0	0	0	0	0	1	0	0	RdHi	RdLo	Rs	1	0	0	1	Rm
Multiply (acc) long	cond	0	0	0	0	1	Un	A	S	RdHi	RdLo	Rs	1	0	0	1	Rm
Swap/swap byte	cond	0	0	0	1	0	B	0	0	Rn	Rd	SBZ	1	0	0	1	Rm
Load/store register exclusive [3]	cond	0	0	0	1	1	0	0	L	Rn	Rd	SBO	1	0	0	1	SBO
Load/store halfword register offset [1]	cond	0	0	0	P	U	0	W	L	Rn	Rd	SBZ	1	0	1	1	Rm
Load/store halfword immediate offset [1]	cond	0	0	0	P	U	1	W	L	Rn	Rd	HiOffset	1	0	1	1	LoOffset
Load signed halfword/byte immediate offset [1]	cond	0	0	0	P	U	1	W	1	Rn	Rd	HiOffset	1	1	H	1	LoOffset
Load signed halfword/byte register offset [1]	cond	0	0	0	P	U	0	W	1	Rn	Rd	SBZ	1	1	H	1	Rm
Load/store doubleword register offset [2]	cond	0	0	0	P	U	0	W	0	Rn	Rd	SBZ	1	1	St	1	Rm
Load/store doubleword immediate offset [2]	cond	0	0	0	P	U	1	W	0	Rn	Rd	HiOffset	1	1	St	1	LoOffset

Figure A3-3 Multiplies and extra load/store instructions

1. UNPREDICTABLE prior to ARMv4.
2. UNPREDICTABLE prior to ARMv5.
3. UNDEFINED in architecture 5, UNPREDICTABLE prior to ARMv5.

A	Accumulate
Un	1 = Unsigned, 0 = Signed
S	Status register update (SPSR => CPSR)
B	1 = Byte, 0 = Word
P, U, I, W	Pre/post indexing or offset, Up/down, Immediate/register offset, and address Write-back fields for the address mode. See Chapter A5 *ARM Addressing Modes* for more details.
L	1 = Load, 0 = Store
H	1= Halfword, 0 = Byte
St	1 = Store, 0 = Load

——— **Note** ———

Any instruction with bits[27:25] = 000, bit[7] = 1, bit[4] = 1, and cond not equal to 1111, and which is not specified in Figure A3-3 or its notes, is an undefined instruction (or UNPREDICTABLE prior to ARMv4).

A3.1.3 Miscellaneous instructions - split this lot into two sections?

Figure A3-4 and Figure A3-5 on page A3-6 show the remaining ARM instruction encodings. An entry in square brackets, for example [1], indicates that more information is given below the figure.

	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
Move status register to register	cond	0 0 0 1 0	R	0 0	SBO	Rd	SBZ	0 0 0 0	SBZ
Move register to status register	cond	0 0 0 1 0	R	1 0	mask	SBO	SBZ	0 0 0 0	Rm
Branch/exchange instruction set Thumb [1]	cond	0 0 0 1 0	0	1 0	SBO	SBO	SBO	0 0 0 1	Rm
Branch/exchange instruction set Java [5]	cond	0 0 0 1 0	0	1 0	SBO	SBO	SBO	0 0 1 0	Rm
Count leading zeros [2]	cond	0 0 0 1 0	1	1 0	SBO	Rd	SBO	0 0 0 1	Rm
Branch and link/exchange instruction set Thumb [2]	cond	0 0 0 1 0	0	1 0	SBO	SBO	SBO	0 0 1 1	Rm
Saturating add/subtract [4]	cond	0 0 0 1 0	op	0	Rn	Rd	SBZ	0 1 0 1	Rm
Software breakpoint [2,3]	cond	0 0 0 1 0	0	1 0	immed			0 1 1 1	immed
Signed multiplies (type 2) [4]	cond	0 0 0 1 0	op	0	Rd	Rn	Rs	1 y x 0	Rm

Figure A3-4 Miscellaneous instructions

1. Defined in ARMv5 and above, and in T variants of ARMv 4.
2. UNDEFINED in ARMv4, UNPREDICTABLE prior to ARMv4.
3. If the `cond` field of this instruction is not 1110, it is UNPREDICTABLE.
4. Defined in ARMv6 and above, and in E variants of ARMv5. UNDEFINED in ARMv4 and non-E variants of ARMv5, UNPREDICTABLE prior to ARMv4.
5. Defined in ARMv6 and above, and in J variants of ARMv5. UNDEFINED in ARMv4 and non-J variants of ARMv5, UNPREDICTABLE prior to ARMv4.

	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
Parallel add/subtract	cond	0 1 1 0 0	opc1	Rn	Rd	SBO	opc2	1	Rm
Halfword pack	cond	0 1 1 0 1 0 0 0	Rn	Rd	shift_imm	op 0	1	Rm
Word saturate	cond	0 1 1 0 1 U 1	sat_imm	Rd	shift_imm	sh 0	1	Rm
Parallel halfword saturate	cond	0 1 1 0 1 U 1 0	sat_imm	Rd	SBO	0 0 1	1	Rm
Select bytes	cond	0 1 1 0 1 0 0 0	Rn	Rd	SBO	1 0 1	1	Rm
Sign/zero extend (add)	cond	0 1 1 0 1	op	Rn	Rd	rotate SBZ	0 1 1	1	Rm
Multiplies (type 3)	cond	0 1 1 1 0	opc1	Rd/RdHi	Rn/RdLo	Rs	opc2	1	Rm
Unsigned sum of absolute differences	cond	0 1 1 1 1 0 0 0	Rd	Rn*	Rs	0 0 0	1	Rm
Unsigned sum of absolute differences, acc	cond	0 1 1 1 1 0 0 0	Rd	1 1 1 1	Rs	0 0 0	1	Rm

Figure A3-5 Miscellaneous instructions, ARMv6 additions

Rn*   Rn != R15.

The instructions shown in Figure A3-5 are defined in ARMv6 and above. They are UNDEFINED in ARMv4 and non-J variants of ARMv5, and UNPREDICTABLE prior to ARMv4.

— **Note** —

Any instruction with bits[27:23] = 00010, bit[20] = 0, bit[7] and bit[4] not both 1, and cond is not equal to 1111, and which is not specified in Figure A3-4 on page A3-5 or its notes, is UNDEFINED (or UNPREDICTABLE prior to ARMv4).

A3.2   The condition field

Most ARM instructions can be *conditionally executed*, which means that they only have their normal effect on the programmer's model state, memory and coprocessors if the N, Z, C and V flags in the CPSR 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 interrupts and Prefetch Aborts, but has no other effect.

Prior to ARM architecture version 5, all ARM instructions could be conditionally executed. A few instructions have been introduced subsequently which can only be executed unconditionally. See *Unconditional instructions* on page A3-3 for details.

Every instruction contains a 4-bit condition code field in bits 31 to 28:

31	28	27		0
cond				
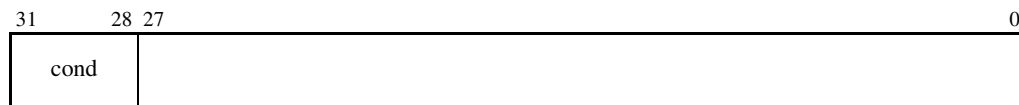
This field contains one of the 16 values described in Table A3-1 on page A3-8. Most instruction mnemonics can be extended with the letters defined in the mnemonic extension field.

If the *always* (AL) condition is specified, the instruction is executed irrespective of the value of the condition code flags. The absence of a condition code on an instruction mnemonic implies the AL condition code.

A3.2.1   Condition code 0b1111

If the condition field is 0b1111, the behavior depends on the architecture version:

*   Prior to ARM architecture version 3, a condition field of 0b1111 meant that the instruction was never executed. The mnemonic extension for this condition was NV.

    ——— **Note** ———

    Use of this condition is now obsolete and unsupported.

*   In ARM architecture version 3 and version 4, any instruction with a condition field of 0b1111 is UNPREDICTABLE.

*   In ARM architecture version 5 and above, a condition field of 0b1111 is used to encode various additional instructions which can only be executed unconditionally (see *Unconditional instructions* on page A3-3). All instruction encoding diagrams which show bits[31:28] as cond only match instructions in which these bits are not equal to 0b1111.

Table A3-1 Condition codes

Opcode [31:28]	Mnemonic extension	Meaning	Condition flag state
0000	EQ	Equal	Z set
0001	NE	Not equal	Z clear
0010	CS/HS	Carry set/unsigned higher or same	C set
0011	CC/LO	Carry clear/unsigned lower	C clear
0100	MI	Minus/negative	N set
0101	PL	Plus/positive or zero	N clear
0110	VS	Overflow	V set
0111	VC	No overflow	V clear
1000	HI	Unsigned higher	C set and Z clear
1001	LS	Unsigned lower or same	C clear or Z set
1010	GE	Signed greater than or equal	N set and V set, or N clear and V clear (N == V)
1011	LT	Signed less than	N set and V clear, or N clear and V set (N != V)
1100	GT	Signed greater than	Z clear, and either N set and V set, or N clear and V clear (Z == 0,N == V)
1101	LE	Signed less than or equal	Z set, or N set and V clear, or N clear and V set (Z == 1 or N != V)
1110	AL	Always (unconditional)	-
1111		See *Condition code 0b1111* on page A3-7	-

ARM DDI 0100F-02

A3.3   Branch instructions

All ARM processors support a branch instruction that allows a conditional branch forwards or backwards up to 32MB. As the PC is one of the general-purpose registers (R15), a branch or jump can also be generated by writing a value to R15.

A subroutine call can be performed by a variant of the standard branch instruction. As well as allowing a branch forward or backward up to 32MB, the Branch with Link (BL) instruction preserves the address of the instruction after the branch (the return address) in the LR (R14).

In T variants of ARM architecture version 4, and in ARM architecture version 5 and above, the Branch and Exchange (BX) instruction copies the contents of a general-purpose register Rm to the PC (like a MOV PC, Rm instruction), with the additional functionality that if bit[0] of the transferred value is 1, the processor shifts to Thumb state. Together with the corresponding Thumb instructions, this allows *interworking* branches between ARM and Thumb code.

Interworking subroutine calls can be generated by combining BX with an instruction to write a suitable return address to the LR, such as an immediately preceding MOV LR, PC instruction.

In ARM architecture version 5 and above, there are also two types of Branch with Link and Exchange (BLX) instruction:

*   One type takes a register operand Rm, like a BX instruction. This instruction behaves like a BX instruction, and additionally writes the address of the next instruction into the LR. This provides a more efficient interworking subroutine call than a sequence of MOV LR, PC followed by BX Rm.

*   The other type behaves like a BL instruction, branching backwards or forwards by up to 32MB and writing a return link to the LR, but shifts to Thumb state rather than staying in ARM state as BL does. This provides a more efficient alternative to loading the subroutine address into Rm followed by a BLX Rm instruction when it is known that a Thumb subroutine is being called and that the subroutine lies within the 32MB range.

A load instruction provides a way to branch anywhere in the 4GB address space (known as a *long branch*). A 32-bit value is loaded directly from memory into the PC, causing a branch. A long branch can be preceded by MOV LR, PC or another instruction that writes the LR to generate a long subroutine call. In ARM architecture version 5 and above, bit[0] of the value loaded by a long branch controls whether the subroutine is executed in ARM state or Thumb state, just like bit[0] of the value moved to the PC by a BX instruction. Prior to ARM architecture version 5, bits[1:0] of the value loaded into the PC are ignored, and a load into the PC can only be used to call a subroutine in ARM state.

In non-T variants of ARM architecture version 5, the instructions described above can cause an entry into Thumb state despite the fact that the Thumb instruction set is not present. This causes the instruction at the branch target to enter the undefined instruction exception. See *The control bits* on page A2-13 for more details.

In ARMv6 and above, and in J variants of ARMv5, there is an additional Branch and Exchange Java instruction, see *BXJ* on page A4-20.

A3.3.1   Examples

```
    B     label               ; branch unconditionally to label

    BCC   label               ; branch to label if carry flag is clear

    BEQ   label               ; branch to label if zero flag is set

    MOV   PC, #0              ; R15 = 0, branch to location zero

    BL    func                ; subroutine call to function


func    .
        .
    MOV   PC, LR               ; R15=R14, return to instruction after the BL
    MOV   LR, PC               ; store the address of the instruction
                               ; after the next one into R14 ready to return
    LDR   PC, =func            ; load a 32-bit value into the program counter
```

A3.3.2   List of branch instructions

B, BL   Branch, and Branch with Link. See *B, BL* on page A4-10.

BLX   Branch with Link and Exchange. See *BLX (1)* on page A4-16 and *BLX (2)* on page A4-18.

BX   Branch and Exchange Instruction Set. See *BX* on page A4-19.

BXJ   Branch and change to Java state. See *BXJ* on page A4-20.

A3.4 Data-processing instructions

ARM has 16 data-processing instructions, shown in Table A3-2.

Table A3-2 Data-processing instructions

Opcode	Mnemonic	Operation	Action
0000	AND	Logical AND	Rd := Rn AND shifter_operand
0001	EOR	Logical Exclusive OR	Rd := Rn EOR shifter_operand
0010	SUB	Subtract	Rd := Rn - shifter_operand
0011	RSB	Reverse Subtract	Rd := shifter_operand - Rn
0100	ADD	Add	Rd := Rn + shifter_operand
0101	ADC	Add with Carry	Rd := Rn + shifter_operand + Carry Flag
0110	SBC	Subtract with Carry	Rd := Rn - shifter_operand - NOT(Carry Flag)
0111	RSC	Reverse Subtract with Carry	Rd := shifter_operand - Rn - NOT(Carry Flag)
1000	TST	Test	Update flags after Rn AND shifter_operand
1001	TEQ	Test Equivalence	Update flags after Rn EOR shifter_operand
1010	CMP	Compare	Update flags after Rn - shifter_operand
1011	CMN	Compare Negated	Update flags after Rn + shifter_operand
1100	ORR	Logical (inclusive) OR	Rd := Rn OR shifter_operand
1101	MOV	Move	Rd := shifter_operand (no first operand)
1110	BIC	Bit Clear	Rd := Rn AND NOT(shifter_operand)
1111	MVN	Move Not	Rd := NOT shifter_operand (no first operand)

Most data-processing instructions take two source operands, though Move and Move Not take only one. The compare and test instructions only update the condition flags. Other data-processing instructions store a result to a register and optionally update the condition flags as well.

Of the two source operands, one is always a register. The other is called a *shifter operand* and is either an immediate value or a register. If the second operand is a register value, it can have a shift applied to it.

CMP, CMN, TST and TEQ always update the condition code flags. The assembler automatically sets the S bit in the instruction for them, and the corresponding instruction with the S bit clear is not a data-processing instruction, but instead lies in one of the instruction extension spaces (see *Extending the instruction set* on page A3-35). The remaining instructions update the flags if an S is appended to the instruction mnemonic (which sets the S bit in the instruction). See *The condition code flags* on page A2-10 for more details.

A3.4.1   Instruction encoding

```
<opcode1>{<cond>}{S}  <Rd>, <shifter_operand>
<opcode1> := MOV | MVN
<opcode2>{<cond>}  <Rn>, <shifter_operand>
<opcode2> := CMP | CMN | TST | TEQ
<opcode3>{<cond>}{S}  <Rd>, <Rn>, <shifter_operand>
<opcode3> := ADD | SUB | RSB | ADC | SBC | RSC | AND | BIC | EOR | ORR
```

31 28	27 26	25	24 21	20	19 16	15 12	11 0
cond	0 0	I	opcode	S	Rn	Rd	shifter_operand
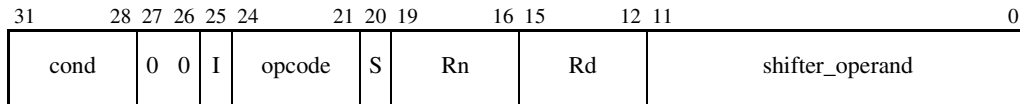
I bit	Distinguishes between the immediate and register forms of `<shifter_operand>`.
S bit	Signifies that the instruction updates the condition codes.
Rn	Specifies the first source operand register.
Rd	Specifies the destination register.
shifter_operand	Specifies the second source operand. See *Addressing Mode 1 - Data-processing operands* on page A5-2 for details of the shifter operands.

A3.4.2 List of data-processing instructions

ADC   Add with Carry. See *ADC* on page A4-4.

ADD   Add. See *ADD* on page A4-6.

AND   Logical AND. See *AND* on page A4-8.

BIC   Logical Bit Clear. See *BIC* on page A4-12.

CMN   Compare Negative. See *CMN* on page A4-25.

CMP   Compare. See *CMP* on page A4-27.

EOR   Logical EOR. See *EOR* on page A4-31.

MOV   Move. See *MOV* on page A4-67.

MVN   Move Negative. See *MVN* on page A4-81.

ORR   Logical OR. See *ORR* on page A4-83.

RSB   Reverse Subtract. See *RSB* on page A4-116.

RSC   Reverse Subtract with Carry. See *RSC* on page A4-118.

SBC   Subtract with Carry. See *SBC* on page A4-132.

SUB   Subtract. See *SUB* on page A4-214.

TEQ   Test Equivalence. See *TEQ* on page A4-228.

TST   Test. See *TST* on page A4-230.

A3.5   Multiply instructions

ARM has several classes of Multiply instruction:

Normal   32-bit ∞ 32-bit, bottom 32-bit result

Long   32-bit ∞ 32-bit, 64-bit result

Halfword   16-bit ∞ 16-bit, 32-bit result

Word ∞ halfword   32-bit ∞ 16-bit, top 32-bit result

Most significant word

                    32-bit ∞ 32-bit, top 32-bit result

Dual halfword   dual 16-bit ∞ 16-bit, 32-bit result.

All Multiply instructions take two register operands as the input to the multiplier. The ARM processor does not directly support a multiply-by-constant instruction because of the efficiency of shift and add, or shift and reverse subtract instructions.

A3.5.1   Normal multiply

There are two 32-bit ∞ 32-bit Multiply instructions that produce bottom 32-bit results:

MUL   Multiplies the values of two registers together, truncates the result to 32 bits, and stores the result in a third register.

MLA   Multiplies the values of two registers together, adds the value of a third register, truncates the result to 32 bits, and stores the result in a fourth register. This can be used to perform multiply-accumulate operations.

Both Normal Multiply instructions can optionally set the N (Negative) and Z (Zero) condition code flags. No distinction is made between signed and unsigned variants. Only the least significant 32 bits of the result are stored in the destination register, and the sign of the operands does not affect this value.

These instructions are available in architecture v2 and above.

A3.5.2   Long multiply

There are five 32-bit ∞ 32-bit Multiply instructions that produce 64-bit results.

Two of the variants multiply the values of two registers together and store the 64-bit result in third and fourth registers. There are signed (SMULL) and unsigned (UMULL) variants. The signed variants produce a different result in the most significant 32 bits if either or both of the source operands is negative.

Two variants multiply the values of two registers together, add the 64-bit value from the third and fourth registers, and store the 64-bit result back into those registers (third and fourth). There are signed (SMLAL) and unsigned (UMLAL) variants. These instructions perform a long multiply and accumulate.

UMAAL multiplies the unsigned values of two registers together, adds the two unsigned 32-bit values from the third and fourth registers, and stores the 64-bit unsigned result back into those registers (third and fourth).

All the Long Multiply instructions except UMAAL can optionally set the N (Negative) and Z (Zero) condition code flags. UMAAL does not affect any flags.

All these instructions except UMAAL are available in ARMv3M, and ARMv4 and above. UMAAL is available in ARMv6 and above.

A3.5.3   Halfword multiply

There are three signed 16-bit ∞ 16-bit Multiply instructions that produce 32-bit results:

SMULxy   Multiplies the 16-bit values of two half-registers together, and stores the signed 32-bit result in a third register.

SMLAxy   Multiplies the 16-bit values of two half-registers together, adds the 32-bit value from a third register, and stores the signed 32-bit result in a fourth register.

SMLALxy   Multiplies the 16-bit values of two half-registers together, adds the 64-bit value from a third and fourth register, and stores the 64-bit result back into those registers (third and fourth).

These instructions do not affect any flags.

They are available in ARMv5E, and ARMv6 and above.

A3.5.4   Word ∞ halfword multiply

There are two signed Multiply instructions that produce top 32-bit results:

SMULWy   Multiplies the 32-bit value of one register with the 16-bit value of either halfword of a second register, and stores the top 32 bits of the signed 48-bit result in a third register.

SMLAWy   Multiplies the 32-bit value of one register with the 16-bit value of either halfword of a second register, extracts the top 32 bits, adds the 32-bit value from a third register, and stores the signed 32-bit result in a fourth register.

SMLAWy sets the Q flag if overflow occurs in the multiplication. SMULWy does not affect any flags.

These instructions are available in ARMv5E, and ARMv6 and above.

A3.5.5   Most significant word multiply

There are three signed 32-bit ∞ 32-bit Multiply instructions that produce top 32-bit results:

SMMUL   Multiplies the 32-bit values of two registers together, and stores the top 32 bits of the signed 64-bit result in a third register.

SMMLA   Multiplies the 32-bit values of two registers together, extracts the top 32 bits, adds the 32-bit value from a third register, and stores the signed 32-bit result in a fourth register.

SMMLS   Multiplies the 32-bit value of two registers together, extracts the top 32 bits, subtracts this
               from a 32-bit value from a third register, and stores the signed 32-bit result in a fourth
               register.

These instructions do not affect any flags.

They are available in ARMv6 and above.

A3.5.6   Dual halfword multiply

There are six dual, signed 16-bit ∞ 16-bit Multiply instructions:

SMUAD   Multiplies the values of the top halfwords of two registers together, multiplies the values of
               the bottom halfwords of the same two registers together, adds the products, and stores the
               32-bit result in a third register.

SMUSD   Multiplies the values of the top halfwords of two registers together, multiplies the values of
               the bottom halfwords of the same two registers together, subtracts one products from the
               other, and stores the 32-bit result in a third register.

SMLAD   Multiplies the 32-bit value of two registers together, extracts the top 32 bits, subtracts this
               from a 32-bit value from a third register, and stores the signed 32-bit result in a fourth
               register.

SMLSD   Multiplies the 32-bit values of two registers together, extracts the top 32 bits, adds the 32-bit
               value from a third register, and stores the signed 32-bit result in a fourth register.

SMLALD   Multiplies the 32-bit value of two registers together, extracts the top 32 bits, subtracts this
               from a 32-bit value from a third register, and stores the signed 32-bit result in a fourth
               register.

SMLSLD   Multiplies the 32-bit value of two registers together, extracts the top 32 bits, subtracts this
               from a 32-bit value from a third register, and stores the signed 32-bit result in a fourth
               register.

These instructions do not affect any flags.

They are available in ARMv6 and above.

A3.5.7   Examples

```
MUL    R4, R2, R1                    ; Set R4 to value of R2 multiplied by R1
MULS   R4, R2, R1                    ; R4 = R2 x R1, set N and Z flags
MLA    R7, R8, R9, R3                ; R7 = R8 x R9 + R3
SMULL  R4, R8, R2, R3                ; R4 = bits 0 to 31 of R2 x R3
                                     ; R8 = bits 32 to 63 of R2 x R3
UMULL  R6, R8, R0, R1                ; R8, R6 = R0 x R1
UMLAL  R5, R8, R0, R1                ; R8, R5 = R0 x R1 + R8, R5
```

ARM DDI 0100F-02

A3.5.8   List of multiply instructions

MLA	Multiply Accumulate. See *MLA* on page A4-65.
MUL	Multiply. See *MUL* on page A4-79.
SMLA<x><y>	
	Signed halfword Multiply Accumulate. See *SMLA<x><y>* on page A4-149.
SMLAD	Signed halfword Multiply Accumulate, Dual. See *SMLAD* on page A4-152.
SMLAL	Signed Multiply Accumulate Long. See *SMLAL* on page A4-154.
SMLAL<x><y>	
	Signed halfword Multiply Accumulate Long. See *SMLAL<x><y>* on page A4-156.
SMLALD	Signed halfword Multiply Accumulate Long, Dual. See *SMLALD* on page A4-158.
SMLAW<y>	Signed halfword by word Multiply Accumulate. See *SMLAW<y>* on page A4-160.
SMLSD	Signed halfword Multiply Subtract, Dual. See *SMLAD* on page A4-152.
SMLSLD	Signed halfword Multiply Subtract Long Dual. See *SMLALD* on page A4-158.
SMMLA	Signed Most significant word Multiply Accumulate. See *SMMLA* on page A4-166.
SMMLS	Signed Most significant word Multiply Subtract. See *SMMLA* on page A4-166.
SMMUL	Signed Most significant word Multiply. See *SMMUL* on page A4-170.
SMUAD	Signed halfword Multiply, Add, Dual. See *SMUAD* on page A4-172.
SMUL<x><y>	
	Signed halfword Multiply. See *SMUL<x><y>* on page A4-174.
SMULL	Signed Multiply Long. See *SMULL* on page A4-176.
SMULW<y>	Signed halfword by word Multiply. See *SMULW<y>* on page A4-178.
SMUSD	Signed halfword Multiply, Subtract, Dual. See *SMUSD* on page A4-180.
UMAAL	Unsigned Multiply Accumulate significant Long. See *UMAAL* on page A4-256.
UMLAL	Unsigned Multiply Accumulate Long. See *UMLAL* on page A4-258.
UMULL	Unsigned Multiply Long. See *UMULL* on page A4-260.

A3.6 Parallel addition and subtraction instructions

In addition to the normal data-processing and multiply instructions, ARMv6 and above include a set of parallel addition and subtraction instructions.

There are six basic instructions:

ADD16   Adds the top halfwords of two registers to form the top halfword of the result.

Adds the bottom halfwords of the same two registers to form the bottom halfword of the result.

ADDSUBX   Does the following:
1.   Exchanges halfwords of the second operand register.
2.   Adds top halfwords and subtracts bottom halfwords.

SUBADDX   Does the following:
1.   Exchanges halfwords of the second operand register.
2.   Subtracts top halfwords and adds bottom halfwords.

SUB16   Subtracts the top halfword of the first operand register from the top halfword of the second operand register to form the top halfword of the result.

Subtracts the bottom halfword of the second operand registers from the bottom halfword of the first operand register to form the bottom halfword of the result.

ADD8   Adds each byte of the second operand register to the corresponding byte of the first operand register to form the corresponding byte of the result.

SUB8   Subtracts each byte of the second operand register from the corresponding byte of the first operand register to form the corresponding byte of the result.

Each of the six instructions is available in the following variations, indicated by the prefixes shown:

S   Signed arithmetic modulo 2^8 or 2^{16}. Sets the CPSR GE bits (see *The GE[3:0] bits* on page A2-12).

Q   Signed saturating arithmetic.

SH   Signed arithmetic, halving the results to avoid overflow.

U   Unsigned arithmetic modulo 2^8 or 2^{16}. Sets the CPSR GE bits (see *The GE[3:0] bits* on page A2-12).

UQ   Unsigned saturating arithmetic.

UH   Unsigned arithmetic, halving the results to avoid overflow.

A3.6.1 List of parallel arithmetic instructions

QADD16	Dual 16-bit signed saturating addition. See *QADD16* on page A4-93.
QADD8	Quad 8-bit signed saturating addition. See *QADD8* on page A4-95.
QADDSUBX	16-bit exchange, signed saturating addition, subtraction. See *QADDSUBX* on page A4-97.
QSUB16	Dual 16-bit signed saturating subtraction. See *QSUB16* on page A4-104.
QSUB8	Quad 8-bit signed saturating subtraction. See *QSUB8* on page A4-106.
QSUBADDX	16-bit exchange, signed saturating subtraction, addition. See *QSUBADDX* on page A4-108.
SADD16	Dual 16-bit signed addition. See *SADD16* on page A4-120.
SADD8	Quad 8-bit signed addition. See *SADD8* on page A4-124.
SADDSUBX	16-bit exchange, signed addition, subtraction. See *SADDSUBX* on page A4-130.
SSUB16	Dual 16-bit signed subtraction. See *SSUB16* on page A4-188.
SSUB8	Quad 8-bit signed subtraction. See *SSUB8* on page A4-190.
SSUBADDX	16-bit exchange, signed subtraction, addition. See *SSUBADDX* on page A4-192.
SHADD16	Dual 16-bit signed half addition. See *SHADD16* on page A4-137.
SHADD8	Quad 8-bit signed half addition. See *SHADD8* on page A4-139.
SHADDSUBX	16-bit exchange, signed half addition, subtraction. See *SHADDSUBX* on page A4-141.
SHSUB16	Dual 16-bit signed half subtraction. See *SHSUB16* on page A4-143.
SHSUB8	Quad 8-bit signed half subtraction. See *SHSUB8* on page A4-145.
SHSUBADDX	16-bit exchange, signed half subtraction, addition. See *SHSUBADDX* on page A4-147.
UADD16	Dual 16-bit unsigned addition. See *UADD16* on page A4-232.
UADD8	Quad 8-bit unsigned addition. See *UADD8* on page A4-236.
UADDSUBX	16-bit exchange, unsigned addition, subtraction. See *UADDSUBX* on page A4-242.
USUB16	Dual 16-bit unsigned subtraction. See *USUB16* on page A4-279.
USUB8	Quad 8-bit unsigned subtraction. See *USUB8* on page A4-280.
USUBADDX	16-bit exchange, unsigned subtraction, addition. See *USUBADDX* on page A4-282.
UHADD16	Dual 16-bit unsigned half addition. See *UHADD16* on page A4-244.
UHADD8	Quad 8-bit unsigned half addition. See *UHADD8* on page A4-246.
UHADDSUBX	16-bit exchange, unsigned half addition, subtraction. See *UHADDSUBX* on page A4-248.
UHSUB16	Dual 16-bit unsigned half subtraction. See *UHSUB16* on page A4-250.
UHSUB8	Quad 8-bit unsigned half subtraction. See *UHSUB16* on page A4-250.
UHSUBADDX	16-bit exchange, unsigned half subtraction, addition. See *UHSUBADDX* on page A4-254.
UQADD16	Dual 16-bit unsigned saturating addition. See *UQADD16* on page A4-262.
UQADD8	Quad 8-bit unsigned saturating addition. See *UQADD8* on page A4-263.
UQADDSUBX	16-bit exchange, unsigned saturating addition, subtraction. See *UQADDSUBX* on page A4-265.
UQSUB16	Dual 16-bit unsigned saturating subtraction. See *UQSUB16* on page A4-266.
UQSUB8	Quad 8-bit unsigned saturating subtraction. See *UQSUB8* on page A4-267.
UQSUBADDX	16-bit exchange, unsigned saturating subtraction, addition. See *UQSUBADDX* on page A4-269.

A3.7 Unpacking instructions

ARM architecture version 6 and above provide several instructions for unpacking data by sign or zero extending bytes to halfwords or words, and halfwords to words. You can optionally add the result to the contents of another register. You can rotate the operand register by any multiple of 8 bits before extending.

There are six basic instructions:

ADD8TO16   Sign extend bits[23:16] and bits[7:0] of one register to 16 bits, and add corresponding halfwords to the values in another register.

ADD8TO32   Sign extend bits[7:0] of one register to 32 bits, and add to the value in another register.

ADD16TO32   Sign extend bits[15:0] of one register to 32 bits, and add to the value in another register.

UNPK8TO16   Sign extend bits[23:16] and bits[7:0] to 16 bits each.

UNPK8TO32   Sign extend bits[7:0] to 32 bits.

UNPK16TO32 Sign extend bits[15:0] to 32 bits.

Each of the six instructions is available in the following variations, indicated by the prefixes shown:

S   Sign extension, with or without addition modulo 2^{16} or 2^{32}.

U   Zero extension, with or without addition modulo 2^{16} or 2^{32}.

A3.7.1 List of sign/zero extend and add instructions

SADD8TO16   Sign extend bytes to halfwords, add halfwords. See *SADD8TO16* on page A4-126.

SADD8TO32   Sign extend byte to word, add. See *SADD8TO32* on page A4-128.

SADD16TO32   Sign extend halfword to word, add. See *SADD16TO32* on page A4-122.

SUNPK8TO16   Signed unpack bytes to halfwords. See *SUNPK8TO16* on page A4-218.

SUNPK8TO32   Signed unpack byte to word. See *SUNPK8TO32* on page A4-220.

SUNPK16TO32   Signed unpack halfword to word. See *SUNPK16TO32* on page A4-216.

UADD8TO16   Zero extend bytes to halfwords, add halfwords. See *UADD8TO16* on page A4-238.

UADD8TO32   Zero extend byte to word, add. See *UADD8TO32* on page A4-240.

UADD16TO32   Zero extend halfword to word, add. See *UADD16TO32* on page A4-234.

UUNPK8TO16   Unsigned unpack bytes to halfwords. See *UUNPK8TO16* on page A4-286.

UUNPK8TO32   Unsigned unpack byte to word. See *UUNPK8TO32* on page A4-288.

UUNPK16TO32   Unsigned unpack halfword to word. See *UUNPK16TO32* on page A4-284.

A3.8 Miscellaneous arithmetic instructions

ARMv5 and above include several miscellaneous arithmetic instructions.

A3.8.1 Count leading zeroes

ARMv5 and above include a Count Leading Zeros (CLZ) instruction. This instruction returns the number of 0 bits at the most significant end of its operand before the first 1 bit is encountered (or 32 if its operand is 0). Two typical applications for this are:

• To determine how many bits the operand should be shifted left to *normalize* it, so that its most significant bit is 1. (This can be used in integer division routines.)

• To locate the highest priority bit in a bit mask.

For details see *CLZ* on page A4-24.

A3.8.2 Unsigned sum of absolute differences

ARMv6 includes an Unsigned Sum of Absolute Differences (USAD8) instruction, and an Unsigned Sum of Absolute Differences and Accumulate (USADA8) instruction.

These instructions do the following:
1. Take corresponding bytes from two registers.
2. Find the absolute differences between the unsigned values of each pair of bytes.
3. Sum the four absolute values.
4. Optionally, accumulate the sum of the absolute differences with the value in a third register.

For details see *USAD8* on page A4-271 and *USADA8* on page A4-273.

A3.9 Other miscellaneous instructions

ARMv6 and above provide several other miscellaneous instructions:

PKHBT (Pack Halfword Bottom Top) combines the bottom, least significant, halfword of its first operand with the top (most significant) halfword of its shifted second operand. The shift is a left shift, by any amount from 0 to 31.

See *PKHBT* on page A4-85.

PKHTB (Pack Halfword Top Bottom) combines the top, most significant, halfword of its first operand with the bottom (least significant) halfword of its shifted second operand. The shift is an arithmetic right shift, by any amount from 1 to 32.

See *PKHTB* on page A4-87.

SSAT (Signed Saturate) saturates a signed value to a signed range. You can choose the bit position at which saturation occurs. You can apply a shift to the value before the saturation occurs.

See *SSAT* on page A4-184.

USAT (Unsigned Saturate) saturates a signed value to an unsigned range. You can choose the bit position at which saturation occurs. You can apply a shift to the value before the saturation occurs.

See *USAT* on page A4-275.

SSAT16 Saturates two 16-bit signed values to a signed range. You can choose the bit position at which saturation occurs.

See *SSAT16* on page A4-186.

USAT16 Saturates two signed 16-bit values to an unsigned range. You can choose the bit position at which saturation occurs.

See *USAT16* on page A4-277.

SEL (Select) selects each byte of its result from either its first operand or its second operand, according to the values of the GE flags. The GE flags record the results of parallel additions or subtractions, see *Parallel addition and subtraction instructions* on page A3-18.

See *SEL* on page A4-134.

A3.10 Status register access instructions

There are two instructions for moving the contents of a program status register to or from a general-purpose register. Both the CPSR and SPSR can be accessed.

In addition, in ARMv6, there are several instructions that can write directly to specific bits, or groups of bits, in the CPSR.

TBD FIX FOLLOWING: Each status register is split into four 8-bit fields that can be individually written:

Bits[31:24]   The flags field.

Bits[23:16]   The status field.

Bits[15:8]   The extension field.

Bits[7:0]   The control field.

To date, the ARM architecture does not use the status and extension fields, and three bits are unused in the flags field. The four condition code flags occupy bits[31:28]. In E variants of architecture versions 5 and above, the Q flag occupies bit[27]. See *The Q flag* on page A2-12 for more information on the Q flag. The control field contains two interrupt disable bits, five processor mode bits, and the Thumb bit on ARM architecture version 5 and above and on T variants of ARM architecture version 4 (see *The T bit* on page A2-13).

The unused bits of the status registers might be used in future ARM architectures, and must not be modified by software. Therefore, a read-modify-write strategy must be used to update the value of a status register to ensure future compatibility.

The status registers are readable to allow the read part of the read-modify-write operation, and to allow all processor state to be preserved (for instance, during process context switches).

The status registers are writable to allow the write part of the read-modify-write operation, and allow all processor state to be restored.

A3.10.1 CPSR value

Altering the value of the CPSR has four uses:
*   sets the value of the condition code flags (and of the Q flag when it exists) to a known value
*   enables or disable interrupts
*   changes processor mode (for instance, to initialize stack pointers)
*   changes the endianness of load and store operations.

——— **Note** ———

The T and J bits must not be changed directly by writing to the CPSR, but only via the BX or BXJ instructions, and in the implicit SPSR to CPSR moves in instructions designed for exception return. Attempts to enter or leave Thumb or Java state by directly altering the T or J bits have UNPREDICTABLE consequences.

A3.10.2 Examples

These examples assume that the ARM processor is already in a privileged mode. If the ARM processor starts in User mode, only the flag update has any effect.

```
MRS    R0, CPSR                ; Read the CPSR
BIC    R0, R0, #0xF0000000      ; Clear the N, Z, C and V bits
MSR    CPSR_f, R0              ; Update the flag bits in the CPSR
                               ; N, Z, C and V flags now all clear

MRS    R0, CPSR                ; Read the CPSR
ORR    R0, R0, #0x80           ; Set the interrupt disable bit
MSR    CPSR_c, R0              ; Update the control bits in the CPSR
                               ; interrupts (IRQ) now disabled

MRS    R0, CPSR                ; Read the CPSR
BIC    R0, R0, #0x1F           ; Clear the mode bits
ORR    R0, R0, #0x11           ; Set the mode bits to FIQ mode
MSR    CPSR_c, R0              ; Update the control bits in the CPSR
                               ; now in FIQ mode
```

A3.10.3 List of status register access instructions

MRS   Move PSR to General-purpose Register. See *MRS* on page A4-73.

MSR   Move General-purpose Register to PSR. See *MSR* on page A4-75.

CPS   Change Processor State. Changes one or more of the processor mode and interrupt enable bits of the CPSR, without changing the other CPSR bits. See *CPS* on page A4-28.

SETEND   Modifies the CPSR endianness, E, bit, without changing any other bits in the CPSR. See *SETEND* on page A4-136.

A3.11   Load and store instructions

The ARM architecture supports two broad types of instruction which load or store the value of a single register, or a pair of registers, from or to memory:

*   The first type can load or store a 32-bit word or an 8-bit unsigned byte.

*   The second type can load or store a 16-bit unsigned halfword, and can load and sign extend a 16-bit halfword or an 8-bit byte. This type of instruction is only available in ARM architecture version 4 and above. In ARMv5E andARMv6, it can also load or store a 64-bit doubleword.

A3.11.1 Addressing modes

In both types of instruction, the addressing mode is formed from two parts:
*   the base register
*   the offset.

The base register can be any one of the general-purpose registers (including the PC, which allows PC-relative addressing for position-independent code).

The offset takes one of three formats:

Immediate   The offset is an unsigned number that can be added to or subtracted from the base register. Immediate offset addressing is useful for accessing data elements that are a fixed distance from the start of the data object, such as structure fields, stack offsets and input/output registers.

For the word and unsigned byte instructions, the immediate offset is a 12-bit number. For the halfword and signed byte instructions, it is an 8-bit number.

Register   The offset is a general-purpose register (not the PC), that can be added to or subtracted from the base register. Register offsets are useful for accessing arrays or blocks of data.

Scaled register   The offset is a general-purpose register (not the PC) shifted by an immediate value, then added to or subtracted from the base register. The same shift operations used for data-processing instructions can be used (Logical Shift Left, Logical Shift Right, Arithmetic Shift Right and Rotate Right), but Logical Shift Left is the most useful as it allows an array indexed to be scaled by the size of each array element.

Scaled register offsets are only available for the word and unsigned byte instructions.

As well as the three types of offset, the offset and base register are used in three different ways to form the memory address. The addressing modes are described as follows:

Offset   The base register and offset are added or subtracted to form the memory address.

Pre-indexed   The base register and offset are added or subtracted to form the memory address. The base register is then updated with this new address, to allow automatic indexing through an array or memory block.

Copyright © 1996–2003 ARM Limited. All rights reserved.
Confidential - Draft

Post-indexed   The value of the base register alone is used as the memory address. The base register and offset are added or subtracted and this value is stored back in the base register, to allow automatic indexing through an array or memory block.

A3.11.2 Load and store word or unsigned byte instructions

Load instructions load a single value from memory and write it to a general-purpose register.

Store instructions read a value from a general-purpose register and store it to memory.

Load and Store instructions have a single instruction format:

```
LDR|STR{<cond>}{B}{T} Rd, <addressing_mode>
```

31 28	27 26	25	24	23	22	21	20	19 16	15 12	11 0
cond	0 1	I	P	U	B	W	L	Rn	Rd	addressing_mode_specific
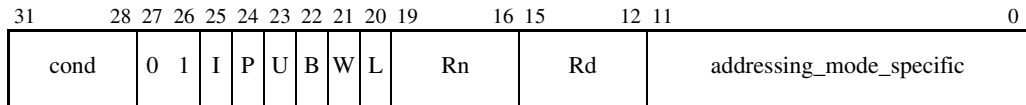
I, P, U, W   Are bits that distinguish between different types of `<addressing_mode>`.

L bit   Distinguishes between a Load (L==1) and a Store instruction (L==0).

B bit   Distinguishes between an unsigned byte (B==1) and a word (B==0) access.

Rn   Specifies the base register used by `<addressing_mode>`.

Rd   Specifies the register whose contents are to be loaded or stored.

A3.11.3 Load and store halfword or doubleword, and load signed byte instructions

Load instructions load a single value from memory and write it to a general-purpose register, or to a pair of general-purpose registers.

Store instructions read a value from a general-purpose register, or from a pair of general-purpose registers, and store it to memory.

These instructions have a single instruction format:

```
LDR|STR{<cond>}H|SH|SB  Rd, <addressing_mode>
```

31 28	27 26 25	24	23	22	21	20	19 16	15 12	11 8	7	6	5	4	3 0
cond	0 0 0	P	U	I	W	L	Rn	Rd	addr_mode	1	S	H	1	addr_mode
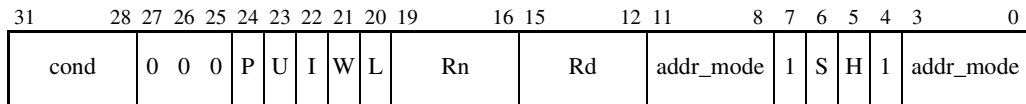
addr_mode   Are addressing-mode-specific bits.

I, P, U, W   Are bits that specify the type of addressing mode (see *Addressing Mode 3 - Miscellaneous Loads and Stores* on page A5-38).

L, S, H   These bits combine to specify signed or unsigned loads or stores, and doubleword, halfword, or byte accesses. See *Addressing Mode 3 - Miscellaneous Loads and Stores* on page A5-38 for details.

Rn   Specifies the base register used by the addressing mode.

Rd   Specifies the register whose contents are to be loaded or stored.

A3.11.4 Examples

```
LDR     R1, [R0]                ; Load R1 from the address in R0
LDR     R8, [R3, #4]            ; Load R8 from the address in R3 + 4
LDR     R12, [R13, #-4]         ; Load R12 from R13 - 4
STR     R2, [R1, #0x100]        ; Store R2 to the address in R1 + 0x100

LDRB    R5, [R9]                ; Load byte into R5 from R9
                                ;  (zero top 3 bytes)
LDRB    R3, [R8, #3]            ; Load byte to R3 from R8 + 3
                                ;  (zero top 3 bytes)
STRB    R4, [R10, #0x200]       ; Store byte from R4 to R10 + 0x200

LDR     R11, [R1, R2]           ; Load R11 from the address in R1 + R2
STRB    R10, [R7, -R4]          ; Store byte from R10 to addr in R7 - R4

LDR     R11, [R3, R5, LSL #2]   ; Load R11 from R3 + (R5 x 4)
LDR     R1, [R0, #4]!           ; Load R1 from R0 + 4, then R0 = R0 + 4
STRB    R7, [R6, #-1]!          ; Store byte from R7 to R6 - 1,
                                ;  then R6 = R6 - 1

LDR     R3, [R9], #4            ; Load R3 from R9, then R9 = R9 + 4
STR     R2, [R5], #8            ; Store R2 to R5, then R5 = R5 + 8

LDR     R0, [PC, #40]           ; Load R0 from PC + 0x40 (= address of
                                ;  the LDR instruction + 8 + 0x40)
LDR     R0, [R1], R2            ; Load R0 from R1, then R1 = R1 + R2

LDRH    R1, [R0]                ; Load halfword to R1 from R0
                                ;  (zero top 2 bytes)
LDRH    R8, [R3, #2]            ; Load halfword into R8 from R3 + 2
LDRH    R12, [R13, #-6]         ; Load halfword into R12 from R13 - 6
STRH    R2, [R1, #0x80]         ; Store halfword from R2 to R1 + 0x80

LDRSH   R5, [R9]                ; Load signed halfword to R5 from R9
LDRSB   R3, [R8, #3]            ; Load signed byte to R3 from R8 + 3
LDRSB   R4, [R10, #0xC1]        ; Load signed byte to R4 from R10 + 0xC1

LDRH    R11, [R1, R2]           ; Load halfword into R11 from address
                                ;  in R1 + R2
STRH    R10, [R7, -R4]          ; Store halfword from R10 to R7 - R4
```

```
        LDRSH   R1, [R0, #2]!            ; Load signed halfword R1 from R0 + 2,
                                         ;  then R0 = R0 + 2

        LDRSB   R7, [R6, #-1]!           ; Load signed byte to R7 from R6 - 1,
                                         ;  then R6 = R6 - 1
        LDRH    R3, [R9], #2             ; Load halfword to R3 from R9,
                                         ;  then R9 = R9 + 2
        STRH    R2, [R5], #8             ; Store halfword from R2 to R5,
                                         ;  then R5 = R5 + 8
```

A3.11.5 List of load and store instructions

LDR   Load Word. See *LDR* on page A4-42.

LDRB   Load Byte. See *LDRB* on page A4-45.

LDRBT   Load Byte with User Mode Privilege. See *LDRBT* on page A4-47.

LDRD   Load Doubleword. See *LDRD* on page A4-49.

LDREX   Load Exclusive. See *LDREX* on page A4-51.

LDRH   Load Unsigned Halfword. See *LDRH* on page A4-53.

LDRSB   Load Signed Byte. See *LDRSB* on page A4-55.

LDRSH   Load Signed Halfword. See *LDRSH* on page A4-57.

LDRT   Load Word with User Mode Privilege. See *LDRT* on page A4-59.

STR   Store Word. See *STR* on page A4-200.

STRB   Store Byte. See *STRB* on page A4-202.

STRBT   Store Byte with User Mode Privilege. See *STRBT* on page A4-204.

STRD   Store Doubleword. See *STRD* on page A4-206.

STREX   Store Exclusive. See *STREX* on page A4-208.

STRH   Store Halfword. See *STRH* on page A4-210.

STRT   Store Word with User Mode Privilege. See *STRT* on page A4-212.

A3.12 Load and Store Multiple instructions

Load Multiple instructions load a subset, or possibly all, of the general-purpose registers from memory.

Store Multiple instructions store a subset, or possibly all, of the general-purpose registers to memory.

Load and Store Multiple instructions have a single instruction format:

```
LDM{<cond>}<addressing_mode>  Rn{!}, <registers>{^}
STM{<cond>}<addressing_mode>  Rn{!}, <registers>{^}
```

where:

```
<addressing_mode> = IA | IB | DA | DB | FD | FA | ED | EA
```

31 28	27 26 25	24	23	22	21	20	19 16	15 0
cond	1 0 0	P	U	S	W	L	Rn	register list

register list   The list of `<registers>` has one bit for each general-purpose register. Bit 0 is for R0, and bit 15 is for R15 (the PC).

    The register syntax list is an opening bracket, followed by a comma-separated list of registers, followed by a closing bracket. A sequence of consecutive registers can be specified by separating the first and last registers in the range with a minus sign.

P, U, and W bits   These distinguish between the different types of addressing mode (see *Addressing Mode 4 - Load and Store Multiple* on page A5-52).

S bit   For `LDM`s that load the PC, the S bit indicates that the CPSR is loaded from the SPSR after all the registers have been loaded. For all `STM`s, and `LDM`s that do not load the PC, it indicates that when the processor is in a privileged mode, the User mode banked registers are transferred and not the registers of the current mode.

L bit   This distinguishes between a Load (L==1) and a Store (L==0) instruction.

Rn   This specifies the base register used by the addressing mode.

A3.12.1 Examples

```
STMFD    R13!, {R0 - R12, LR}
LDMFD    R13!, {R0 - R12, PC}
LDMIA    R0, {R5 - R8}
STMDA    R1!, {R2, R5, R7 - R9, R11}
```

A3.12.2 List of Load and Store Multiple instructions

LDM   Load Multiple. See *LDM (1)* on page A4-35.

LDM   User Registers Load Multiple. See *LDM (2)* on page A4-37.

LDM   Load Multiple with Restore CPSR. See *LDM (3)* on page A4-39.

STM   Store Multiple. See *STM (1)* on page A4-196.

STM   User Registers Store Multiple. See *STM (2)* on page A4-198.

A3.13   Semaphore instructions

The ARM instruction set has two semaphore instructions:

*   Swap (`SWP`)
*   Swap Byte (`SWPB`).

These instructions are provided for process synchronization. Both instructions generate an atomic load and store operation, allowing a memory semaphore to be loaded and altered without interruption.

`SWP` and `SWPB` have a single addressing mode, whose address is the contents of a register. Separate registers are used to specify the value to store and the destination of the load. If the same register is specified for both of these, `SWP` exchanges the value in the register and the value in memory.

The semaphore instructions do not provide a compare and conditional write facility. If wanted, this must be done explicitly.

A3.13.1   Examples

```
SWP    R12, R10, [R9]      ; load R12 from address R9 and
                           ; store R10 to address R9

SWPB   R3, R4, [R8]        ; load byte to R3 from address R8 and
                           ; store byte from R4 to address R8

SWP    R1, R1, [R2]        ; Exchange value in R1 and address in R2
```

A3.13.2   List of semaphore instructions

SWP   Swap. See *SWP* on page A4-224.

SWPB   Swap Byte. See *SWPB* on page A4-226.

A3.14 Exception-generating instructions

The ARM instruction set provides two types of instruction whose main purpose is to cause a processor exception to occur:

- The Software Interrupt (SWI) instruction is used to cause a SWI exception to occur (see *Software Interrupt exception* on page A2-19). This is the main mechanism in the ARM instruction set by which User mode code can make calls to privileged Operating System code.

- The Breakpoint (BKPT) instruction is used for software breakpoints in ARM architecture versions 5 and above. Its default behavior is to cause a Prefetch Abort exception to occur (see *Prefetch Abort (instruction fetch memory abort)* on page A2-20). A debug monitor program which has previously been installed on the Prefetch Abort vector can handle this exception.

  If debug hardware is present in the system, it is allowed to override this default behavior. Details of whether and how this happens are IMPLEMENTATION DEFINED.

A3.14.1 Instruction encodings

```
SWI{<cond>}   <immed_24>
```

31 28	27 26 25 24	23 0
cond	1 1 1 1	immed_24

```
BKPT   <immediate>
```

31 28	27 26 25 24 23 22 21 20	19 8	7 4	3 0
1 1 1 0	0 0 0 1 0 0 1 0	immed	0 1 1 1	immed
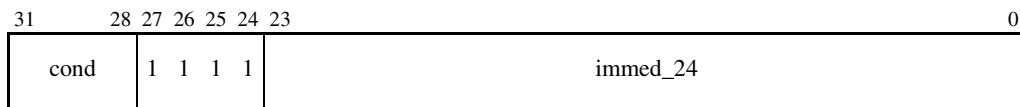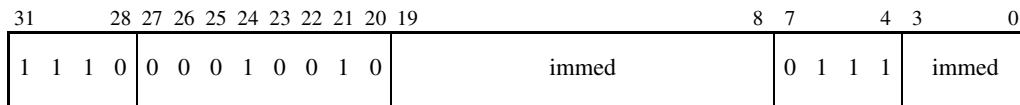
In both SWI and BKPT, the immediate fields of the instruction are ignored by the ARM processor. The SWI or Prefetch Abort handler can optionally be written to load the instruction that caused the exception and extract these fields. This allows them to be used to communicate extra information about the Operating System call or breakpoint to the handler.

A3.14.2 List of exception-generating instructions

BKPT   Breakpoint. See *BKPT* on page A4-14.

SWI   Software Interrupt. See *SWI* on page A4-222.

A3.15 Coprocessor instructions

The ARM instruction set provides three types of instruction for communicating with coprocessors. These allow:

- the ARM processor to initiate a coprocessor data processing operation
- ARM registers to be transferred to and from coprocessor registers
- the ARM processor to generate addresses for the coprocessor Load and Store instructions.

The instruction set distinguishes up to 16 coprocessors with a 4-bit field in each coprocessor instruction, so each coprocessor is assigned a particular number.

——— **Note** ———

One coprocessor can use more than one of the 16 numbers if a large coprocessor instruction set is required.

Coprocessors execute the same instruction stream as ARM, ignoring ARM instructions and coprocessor instructions for other coprocessors. Coprocessor instructions that cannot be executed by coprocessor hardware cause an undefined instruction exception, allowing software emulation of coprocessor hardware.

A coprocessor can partially execute an instruction and then cause an exception. This is useful for handling run-time-generated exceptions, like divide-by-zero or overflow. However, the partial execution is internal to the coprocessor and is not visible to the ARM processor. As far as the ARM processor is concerned, the instruction is held at the start of its execution and completes without exception if allowed to begin execution. Any decision on whether to execute the instruction or cause an exception is taken within the coprocessor before the ARM processor is allowed to start executing the instruction.

Not all fields in coprocessor instructions are used by the ARM processor. Coprocessor register specifiers and opcodes are defined by individual coprocessors. Therefore, only generic instruction mnemonics are provided for coprocessor instructions. Assembler macros can be used to transform custom coprocessor mnemonics into these generic mnemonics, or to regenerate the opcodes manually.

A3.15.1 Examples

```
CDP    p5, 2, c12, c10, c3, 4   ; Coproc 5 data operation
                                 ; opcode 1 = 2, opcode 2 = 4
                                 ; destination register is 12
                                 ; source registers are 10 and 3

MRC    p15, 5, R4, c0, c2, 3     ; Coproc 15 transfer to ARM register
                                 ; opcode 1 = 5, opcode 2 = 3
                                 ; ARM destination register = R4
                                 ; coproc source registers are 0 and 2

MCR    p14, 1, R7, c7, c12, 6    ; ARM register transfer to Coproc 14
                                 ; opcode 1 = 1, opcode 2 = 6
                                 ; ARM source register = R7
                                 ; coproc dest registers are 7 and 12
```

```
        LDC     p6, CR1, [R4]           ; Load from memory to coprocessor 6
                                        ; ARM register 4 contains the address
                                        ; Load to CP reg 1

        LDC     p6, CR4, [R2, #4]       ; Load from memory to coprocessor 6
                                        ; ARM register R2 + 4 is the address
                                        ; Load to CP reg 4

        STC     p8, CR8, [R2, #4]!      ; Store from coprocessor 8 to memory
                                        ; ARM register R2 + 4 is the address
                                        ; after the transfer R2 = R2 + 4
                                        ; Store from CP reg 8

        STC     p8, CR9, [R2], #-16     ; Store from coprocessor 8 to memory
                                        ; ARM register R2 holds the address
                                        ; after the transfer R2 = R2 - 16
                                        ; Store from CP reg 9
```

A3.15.2 List of coprocessor instructions

CDP   Coprocessor Data Operations. See *CDP* on page A4-22.

LDC   Load Coprocessor Register. See *LDC* on page A4-33.

MCR   Move to Coprocessor from ARM Register. See *MCR* on page A4-61.

MCRR   Move to Coprocessor from two ARM Registers. See *MCRR* on page A4-63.

MRC   Move to ARM Register from Coprocessor. See *MRC* on page A4-69.

MRRC   Move to two ARM Registers from Coprocessor. See *MRRC* on page A4-71.

STC   Store Coprocessor Register. See *STC* on page A4-194.

——— **Note** ———

MCRR and MRRC are only available in ARMv5E and ARMv6 and above.

A3.16 Extending the instruction set

Successive versions of the ARM architecture have extended the instruction set in a number of areas. This section describes the six areas where extensions have occurred, and where further extensions might occur in the future:

•   *Undefined instruction extension space* on page A3-36

•   *Arithmetic instruction extension space* on page A3-37

•   *Control and DSP instruction extension space* on page A3-38

•   *Load/store instruction extension space* on page A3-40

•   *Coprocessor instruction extension space* on page A3-41

•   *Unconditional instruction extension space* on page A3-42.

Instructions in these areas which have not yet been allocated a meaning are either UNDEFINED or UNPREDICTABLE. To determine which, use the following rules:

1.   The *decode bits* of an instruction are defined to be bits[27:20] and bits[7:4].

   In ARM architecture version 5 and above, the result of ANDing bits[31:28] together is also a decode bit. This bit determines whether the condition field is 0b1111, which is used in ARM architecture version 5 and above to encode various instructions which can only be executed unconditionally. See *Condition code 0b1111* on page A3-7 and *Unconditional instruction extension space* on page A3-42 for more information.

2.   If the decode bits of an instruction are equal to those of a defined instruction, but the whole instruction is not a defined instruction, then the instruction is UNPREDICTABLE.

   For example, suppose an instruction has:

•   bits[31:28] not equal to 0b1111

•   bits[27:20] equal to 0b00010000

•   bits[7:4] equal to 0b0000

   but where:

•   bit[11] of the instruction is 1.

   Here, the instruction is in the control instruction extension space and has the same decode bits as an MRS instruction, but is not a valid MRS instruction because bit[11] of an MRS instruction should be zero. Using the above rule, this instruction is UNPREDICTABLE.

3.   In ARM architecture version 4 and above, if the decode bits of an instruction are not equal to those of any defined instruction, then the instruction is UNDEFINED.

4.   In ARM architecture version 3 and below, if the decode bits of an instruction are not equal to those of any defined instruction, then the instruction is:

•   UNDEFINED if it is in the undefined instruction space

•   UNPREDICTABLE if it is in any of the other five areas.

Each of rules 2 to 4 above applies separately to each ARM architecture version. As a result, the status of an instruction might differ between architecture versions. Usually, this happens because an instruction which was UNPREDICTABLE or UNDEFINED in an earlier architecture version becomes a defined instruction in a later version.

A3.16.1 Undefined instruction extension space

Instructions with the following opcodes are undefined instruction space:

```
opcode[27:25] = 0b011
opcode[4] = 1
```

31	28	27	26	25	24																				5	4	3			0
cond		0	1	1	op	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	1	x	x	x	x

The meaning of unallocated instructions in the undefined instruction space is UNDEFINED on all versions of the ARM architecture.

In general, undefined instructions might be used to extend the ARM instruction set in the future. However, it is intended that instructions with the following encoding will not be used for this:

31	28	27	26	25	24	23	22	21	20	19												8	7	6	5	4	3	2	1	0
cond		0	1	1	1	1	1	1	1	x	x	x	x	x	x	x	x	x	x	x	x	x	1	1	1	1	x	x	x	x

If a programmer wants to use an undefined instruction for software purposes, with minimal risk that future hardware will treat it as a defined instruction, one of the instructions with this encoding must be used.

Table A3-3 summarizes the instructions that have already been allocated in this area.

Table A3-3 Undefined instruction space

Instructions	op	Architecture versions
Parallel additions, subtractions, and addition with subtractions. See *Parallel addition and subtraction instructions* on page A3-18.	00	ARMv6 and above
PKH, SSAT, SSAT16, USAT, USAT16, SEL Also sign/zero extend and add instructions. See *Unpacking instructions* on page A3-20.	01	ARMv6 and above
SMLAD, SMLSD, SMLALD, SMUAD, SMUSD	10	ARMv6 and above
USAD8, USADA8	11	ARMv6 and above

A3.16.2 Arithmetic instruction extension space

Instructions with the following opcodes are the arithmetic instruction extension space:

```
opcode[27:24]  == 0b0000
opcode[7:4]    == 0b1001
opcode[31:28]  != 0b1111  /* Only required for version 5 and above */
```

The field names given are guidelines suggested to simplify implementation.

31 28	27 26 25 24	23 20	19 16	15 12	11 8	7 6 5 4	3 0
cond	0 0 0 0	op1	Rn	Rd	Rs	1 0 0 1	Rm

Table A3-4 summarizes the instructions that have already been allocated in this area.

Table A3-4 Arithmetic instruction space

Instructions	op1	Architecture versions
MUL, MULS	000S	ARMv2 and above
MLA, MLAS	001S	ARMv2 and above
UMULL, UMULLS	100S	All M variants
UMLAL, UMLALS	101S	All M variants
SMULL, SMULLS	110S	All M variants
SMLAL, SMLALS	111S	All M variants

A3.16.3 Control and DSP instruction extension space

Instructions with the following opcodes are the control instruction space.

```
opcode[27:26] == 0b00
opcode[24:23] == 0b10
opcode[20]    == 0
opcode[31:28] != 0b1111  /* Only required for version 5 and above */
```

and not:

```
opcode[25] == 0
opcode[7]  == 1
opcode[4]  == 1
```

The field names given are guidelines suggested to simplify implementation.

31 28	27 26	25 24 23 22 21	20	19 16	15 12	11 8	7 6 5	4	3 0
cond	0 0	0 1 0 op1	0	Rn	Rd	Rs	op2	0	Rm
cond	0 0	0 1 0 op1	0	Rn	Rd	Rs	0 op2	1	Rm
cond	0 0	1 1 0 op1	0	Rn	Rd	rotate_imm	immed_8		

Table A3-5 summarizes the instructions that have already been allocated in this area.

Table A3-5 Control and DSP extension space instructions

Instruction	Bit[25]	Bits[7:4]	op1	Architecture versions
MRS	0	0000	x0	Version 3 and above
MSR (register form)	0	0000	x1	Version 3 and above
BX	0	0001	01	Version 5 and above, plus T variants of version 4
CLZ	0	0001	11	Version 5 and above
BXJ	0	0010	01	Version 5EJ and 6 and above
BLX (register form)	0	0011	01	Version 5 and above
QADD	0	0101	00	E variants of version 5 and above
QSUB	0	0101	01	E variants of version 5 and above
QDADD	0	0101	10	E variants of version 5 and above

Copyright © 1996–2003 ARM Limited. All rights reserved.   ARM DDI 0100F-02
Confidential - Draft

Table A3-5 Control and DSP extension space instructions (continued)

Instruction	Bit[25]	Bits[7:4]	op1	Architecture versions
QDSUB	0	0101	11	E variants of version 5 and above
BKPT	0	0111	01	Version 5 and above
SMLA<x><y>	0	1yx0	00	E variants of version 5 and above
SMLAW<y>	0	1y00	01	E variants of version 5 and above
SMULW<y>	0	1y10	01	E variants of version 5 and above
SMLAL<x><y>	0	1yx0	10	E variants of version 5 and above
SMUL<x><y>	0	1yx0	11	E variants of version 5 and above
MSR (immediate form)	1	xxxx	x1	Version 3 and above

A3.16.4 Load/store instruction extension space

Instructions with the following opcodes are the load/store instruction extension space:

```
opcode[27:25] == 0b000
opcode[7]     == 1
opcode[4]     == 1
opcode[31:28] != 0b1111 /* Only required for version 5 and above */
```

and not:

```
opcode[24]  == 0
opcode[6:5] == 0
```

The field names given are guidelines suggested to simplify implementation.

31 28	27 26 25	24	23	22	21	20	19 16	15 12	11 8	7	6 5	4	3 0
cond	0 0 0	P	U	B	W	L	Rn	Rd	Rs	1	op1	1	Rm
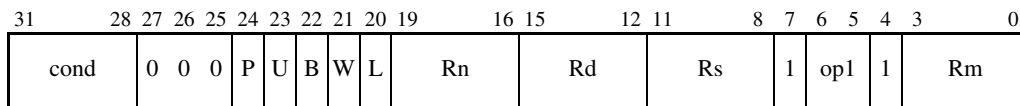
Table A3-6 summarizes the instructions that have already been allocated in this area.

Table A3-6 Load/store instructions

Instruction	Bits[24:20]					op1		Architecture versions
SWP/SWPB	1	0	B	0	0	0	0	Version 3 and above, plus ARMv2a
LDREX	1	1	0	0	1	0	0	Version 6 and above
STREX	1	1	0	0	0	0	0	Version 6 and above
STRH	P	U	I	W	0	0	1	Version 4 and above
LDRD	P	U	I	W	0	1	0	E variants of version 5 and above, except v5TExP
STRD	P	U	I	W	0	1	1	E variants of version 5 and above, except v5TExP
LDRH	P	U	I	W	1	0	1	Version 4 and above
LDRSB	P	U	I	W	1	1	0	Version 4 and above
LDRSH	P	U	I	W	1	1	1	Version 4 and above

A3.16.5 Coprocessor instruction extension space

Instructions with the following opcodes are the coprocessor instruction extension space:

```
opcode[27:23]  == 0b11000
opcode[21]     == 0
```

The field names given are guidelines suggested to simplify implementation.

31	28	27 26 25 24 23	22	21	20	19	16 15	12 11	8 7	0
cond		1 1 0 0 0	x	0	x	Rn	CRd	cp_num	offset	
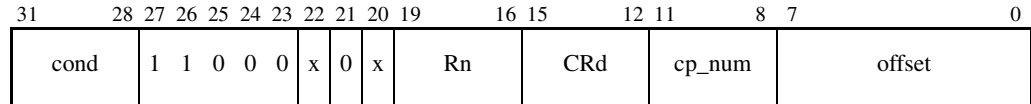
In ARM architecture version 3 and below, all instructions in the coprocessor instruction extension space are UNPREDICTABLE.

In all variants of architecture version 4, and in non-E variants of architecture 5, all instructions in the coprocessor instruction extension space are UNDEFINED. It is IMPLEMENTATION DEFINED how an ARM processor achieves this. The options are:

- The ARM processor might take the undefined instruction exception directly.

- The ARM processor might require attached coprocessors not to respond to such instructions. This causes the undefined instruction exception to be taken (see *Undefined Instruction exception* on page A2-18).

In E variants of architecture version 5, instructions in the coprocessor instruction extension space are treated as follows:

- Instructions with bit[22] == 0 are UNDEFINED and are handled in precisely the same way as described above for non-E variants.

- Instructions with bit[22] ==1 are the MCRR and MRRC instructions, see *MCRR* on page A4-63 and *MRRC* on page A4-71.

A3.16.6 Unconditional instruction extension space

In ARM architecture version 5 and above, instructions with the following opcode are the unconditional instruction space:

```
opcode[31:28] == 0b1111
```

31 30 29 28	27　　　opcode1　　　20	19　x x x x x x x x x x x x x　8	7　opcode2　4	3　x x x x　0
1　1　1　1	opcode1	x x x x x x x x x x x x x	opcode2	x x x x
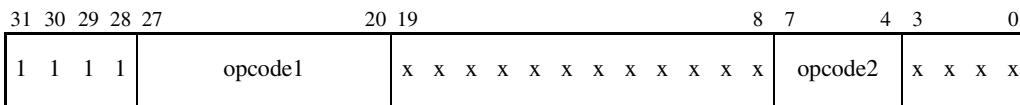
Table A3-7 summarizes the instructions that have already been allocated in this area.

Table A3-7 Unconditional instruction extension space

Instruction	opcode1								opcode2				Architecture versions
CPS/SETEND	0	0	0	1	0	0	0	0	x	x	x	0	Version 6 and above
PLD	0	1	I	1	U	1	0	1	x	x	x	x	E variants of version 5 and above, except v5TExP
BLX (address form)	1	0	1	x	x	x	x	x	x	x	x	x	Version 5 and above
MCRR2	1	1	0	0	0	1	0	0	x	x	x	x	Version 6 and above
MRRC2	1	1	0	0	0	1	0	1	x	x	x	x	Version 6 and above
STC2	1	1	0	x	x	x	x	0	x	x	x	x	Version 5 and above
LDC2	1	1	0	x	x	x	x	1	x	x	x	x	Version 5 and above
CDP2	1	1	1	0	x	x	x	x	x	x	x	0	Version 5 and above
MCR2	1	1	1	0	x	x	x	0	x	x	x	1	Version 5 and above
MRC2	1	1	1	0	x	x	x	1	x	x	x	1	Version 5 and above

Chapter A4
ARM Instructions

This chapter describes the syntax and usage of every ARM instruction, in the sections:

- *Alphabetical list of ARM instructions* on page A4-2
- *ARM instructions and architecture versions* on page A4-290.

A4.1 Alphabetical list of ARM instructions

Every ARM instruction is listed on the following pages. Each instruction description shows:

- the instruction encoding
- the instruction syntax
- the version of the ARM architecture where the instruction is valid
- any exceptions that apply
- an example in pseudo-code of how the instruction operates
- notes on usage and special cases.

A4.1.1 General notes

These notes explain the types of information and abbreviations used on the instruction pages.

Syntax abbreviations

The following abbreviations are used in the instruction pages:

immed_*n*   This is an immediate value, where *n* is the number of bits. For example, an 8-bit immediate value is represented by:

```
immed_8
```

offset_*n*   This is an offset value, where *n* is the number of bits. For example, an 8-bit offset value is represented by:

```
offset_8
```

The same construction is used for signed offsets. For example, an 8-bit signed offset is represented by:

```
signed_offset_8
```

Encoding diagram and assembler syntax

For the conventions used, see *Assembler syntax descriptions* on page xx.

Architecture versions

This gives details of architecture versions where the instruction is valid. For further information on architecture versions, see *Architecture versions and variants* on page xi.

Exceptions

This gives details of which exceptions can occur during the execution of the instruction. Prefetch Abort is not listed in general, both because it can occur for any instruction and because if an abort occurred during instruction fetch, the instruction bit pattern is not known. (Prefetch Abort is however listed for BKPT, since it can generate a Prefetch Abort exception without these considerations applying.)

Operation

This gives a pseudo-code description of what the instruction does. For details of conventions used in this pseudo-code, see *Pseudo-code descriptions of instructions* on page xix.

Information on usage

Usage sections are included where appropriate to supply suggestions and other information about how to use the instruction effectively.

A4.1.2 ADC

31	28	27 26	25	24 23 22 21	20	19	16	15	12	11	0
cond		0 0	I	0 1 0 1	S	Rn		Rd		shifter_operand	
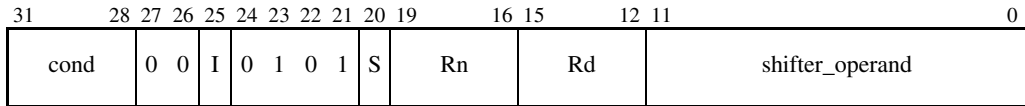
ADC (Add with Carry) adds two values and the Carry flag. The first value comes from a register. The second value can be either an immediate value or a value from a register, and can be shifted before the addition.

ADC can optionally update the condition code flags, based on the result.

Syntax

ADC{<cond>}{S} <Rd>, <Rn>, <shifter_operand>

where:

<cond>   Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-7. If <cond> is omitted, the AL (always) condition is used.

S   Causes the S bit (bit[20]) in the instruction to be set to 1 and specifies that the instruction updates the CPSR. If S is omitted, the S bit is set to 0 and the CPSR is not changed by the instruction. Two types of CPSR update can occur when S is specified:

- If <Rd> is not R15, the N and Z flags are set according to the result of the addition, and the C and V flags are set according to whether the addition generated a carry (unsigned overflow) and a signed overflow, respectively. The rest of the CPSR is unchanged.

- If <Rd> is R15, the SPSR of the current mode is copied to the CPSR. This form of the instruction is UNPREDICTABLE if executed in User mode or System mode, because these modes do not have an SPSR.

<Rd>   Specifies the destination register.

<Rn>   Specifies the register that contains the first operand.

<shifter_operand>

              Specifies the second operand. The options for this operand are described in *Addressing Mode 1 - Data-processing operands* on page A5-2, including how each option causes the I bit (bit[25]) and the shifter_operand bits (bits[11:0]) to be set in the instruction.

              If the I bit is 0 and both bit[7] and bit[4] of shifter_operand are 1, the instruction is not ADC. Instead, see *Extending the instruction set* on page A3-35 to determine which instruction it is.

Architecture version

All.

Exceptions

None.

Operation

```
if ConditionPassed(cond) then
    Rd = Rn + shifter_operand + C Flag
    if S == 1 and Rd == R15 then
        CPSR = SPSR
    else if S == 1 then
        N Flag = Rd[31]
        Z Flag = if Rd == 0 then 1 else 0
        C Flag = CarryFrom(Rn + shifter_operand + C Flag)
        V Flag = OverflowFrom(Rn + shifter_operand + C Flag)
```

Usage

Use ADC to synthesize multi-word addition. If register pairs R0, R1 and R2, R3 hold 64-bit values (where R0 and R2 hold the least significant words) the following instructions leave the 64-bit sum in R4, R5:

```
ADDS R4,R0,R2
ADC  R5,R1,R3
```

If the second instruction is changed from:

```
ADC  R5,R1,R3
```

to:

```
ADCS R5,R1,R3
```

the resulting values of the flags indicate:

N   The 64-bit addition produced a negative result.

C   An unsigned overflow occurred.

V   A signed overflow occurred.

Z   The most significant 32 bits are all zero.

The following instruction produces a single-bit Rotate Left with Extend operation (33-bit rotate through the Carry flag) on R0:

```
ADCS R0,R0,R0
```

See *Data-processing operands - Rotate right with extend* on page A5-18 for information on how to perform a similar rotation to the right.

A4.1.3 ADD

31	28	27	26	25	24	23	22	21	20	19			16	15		12	11		0
cond		0	0	I	0	1	0	0	S	Rn				Rd			shifter operand		
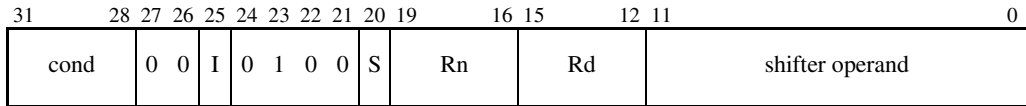
ADD adds two values. The first value comes from a register. The second value can be either an immediate value or a value from a register, and can be shifted before the addition.

ADD can optionally update the condition code flags, based on the result.

Syntax

ADD{<cond>}{S} <Rd>, <Rn>, <shifter_operand>

where:

<cond>   Is the condition under which the instruction is executed. *The condition field* on page A3-7. If <cond> is omitted, the AL (always) condition is used.

S   Causes the S bit (bit[20]) in the instruction to be set to 1 and specifies that the instruction updates the CPSR. If S is omitted, the S bit is set to 0 and the CPSR is not changed by the instruction. Two types of CPSR update can occur when S is specified:

- If <Rd> is not R15, the N and Z flags are set according to the result of the addition, and the C and V flags are set according to whether the addition generated a carry (unsigned overflow) and a signed overflow, respectively. The rest of the CPSR is unchanged.

- If <Rd> is R15, the SPSR of the current mode is copied to the CPSR. This form of the instruction is UNPREDICTABLE if executed in User mode or System mode, because these modes do not have an SPSR.

<Rd>   Specifies the destination register.

<Rn>   Specifies the register that contains the first operand.

<shifter_operand>

             Specifies the second operand. The options for this operand are described in *Addressing Mode 1 - Data-processing operands* on page A5-2, including how each option causes the I bit (bit[25]) and the shifter_operand bits (bits[11:0]) to be set in the instruction.

             If the I bit is 0 and both bit[7] and bit[4] of shifter_operand are 1, the instruction is not ADD. Instead, see *Extending the instruction set* on page A3-35 to determine which instruction it is.

Architecture version

All.

Exceptions

None.

Operation

```
if ConditionPassed(cond) then
    Rd = Rn + shifter_operand
    if S == 1 and Rd == R15 then
        if CurrentModeHasSPSR() then
            CPSR = SPSR
        else UNPREDICTABLE
    else if S == 1 then
        N Flag = Rd[31]
        Z Flag = if Rd == 0 then 1 else 0
        C Flag = CarryFrom(Rn + shifter_operand)
        V Flag = OverflowFrom(Rn + shifter_operand)
```

Usage

Use ADD to add two values together.

To increment a register value in Rx use:

```
ADD Rx, Rx, #1
```

You can perform constant multiplication of Rx by 2^n+1 into Rd with:

```
ADD Rd, Rx, Rx, LSL #n
```

To form a PC-relative address use:

```
ADD Rs, PC, #offset
```

where the offset must be the difference between the required address and the address held in the PC, where the PC is the address of the ADD instruction itself plus 8 bytes.

A4.1.4   AND

31	28	27 26	25	24 23 22 21	20	19	16	15	12	11	0
cond		0 0	I	0 0 0 0	S	Rn		Rd		shifter_operand	

AND performs a bitwise AND of two values. The first value comes from a register. The second value can be either an immediate value or a value from a register, and can be shifted before the AND operation.

AND can optionally update the condition code flags, based on the result.

Syntax

AND{<cond>}{S}   <Rd>, <Rn>, <shifter_operand>

where:

<cond>   Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-7. If <cond> is omitted, the AL (always) condition is used.

S   Causes the S bit (bit[20]) in the instruction to be set to 1 and specifies that the instruction updates the CPSR. If S is omitted, the S bit is set to 0 and the CPSR is not changed by the instruction. Two types of CPSR update can occur when S is specified:

•   If <Rd> is not R15, the N and Z flags are set according to the result of the operation, and the C flag is set to the carry output bit generated by the shifter (see *Addressing Mode 1 - Data-processing operands* on page A5-2). The V flag and the rest of the CPSR are unaffected.

•   If <Rd> is R15, the SPSR of the current mode is copied to the CPSR. This form of the instruction is UNPREDICTABLE if executed in User mode or System mode, because these modes do not have an SPSR.

<Rd>   Specifies the destination register.

<Rn>   Specifies the register that contains the first operand.

<shifter_operand>

Specifies the second operand. The options for this operand are described in *Addressing Mode 1 - Data-processing operands* on page A5-2, including how each option causes the I bit (bit[25]) and the shifter_operand bits (bits[11:0]) to be set in the instruction.

If the I bit is 0 and both bit[7] and bit[4] of shifter_operand are 1, the instruction is not AND. Instead, see *Extending the instruction set* on page A3-35 to determine which instruction it is.

Architecture version

All.

Exceptions

None.

Operation

```
if ConditionPassed(cond) then
    Rd = Rn AND shifter_operand
    if S == 1 and Rd == R15 then
        if CurrentModeHasSPSR() then
            CPSR = SPSR
        else UNPREDICTABLE
    else if S == 1 then
        N Flag = Rd[31]
        Z Flag = if Rd == 0 then 1 else 0
        C Flag = shifter_carry_out
        V Flag = unaffected
```

Usage

AND is most useful for extracting a field from a register, by ANDing the register with a mask value that has 1s in the field to be extracted, and 0s elsewhere.

A4.1.5   B, BL

31	28	27 26 25	24	23	0
cond		1 0 1	L	signed_immed_24	
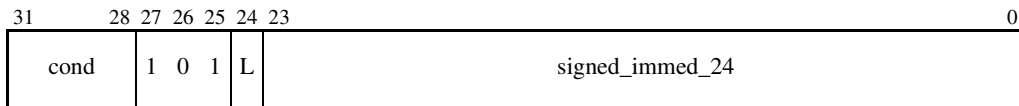
`B` (Branch) and `BL` (Branch and Link) cause a branch to a target address, and provide both conditional and unconditional changes to program flow.

`BL` also stores a return address in the link register, R14 (also known as LR).

Syntax

`B{L}{<cond>}  <target_address>`

where:

L   Causes the `L` bit (bit 24) in the instruction to be set to 1. The resulting instruction stores a return address in the link register (R14). If L is omitted, the L bit is 0 and the instruction simply branches without storing a return address.

`<cond>`   Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-7. If `<cond>` is omitted, the `AL` (always) condition is used.

`<target_address>`

                Specifies the address to branch to. The branch target address is calculated by:

                1.   Sign-extending the 24-bit signed (two's complement) immediate to 32 bits.

                2.   Shifting the result left two bits.

                3.   Adding this to the contents of the PC, which contains the address of the branch instruction plus 8.

                The instruction can therefore specify a branch of approximately ±32MB (see *Usage* on page A4-11 for precise range).

Architecture version

All.

Exceptions

None.

                   ARM DDI 0100F-02

Operation

```
if ConditionPassed(cond) then
    if L == 1 then
        LR = address of the instruction after the branch instruction
    PC = PC + (SignExtend(signed_immed_24) << 2)
```

Usage

Use BL to perform a subroutine call. The return from subroutine is achieved by copying R14 to the PC. Typically, this is done by one of the following methods:

- Executing a BX R14 instruction, on architecture versions that support that instruction.

- Executing a MOV PC,R14 instruction.

- Storing a group of registers and R14 to the stack on subroutine entry, using an instruction of the form:

      STMFD R13!,{<registers>,R14}

   and then restoring the register values and returning with an instruction of the form:

      LDMFD R13!,{<registers>,PC}

To calculate the correct value of signed_immed_24, the assembler (or other toolkit component) must:

1. Form the base address for this branch instruction. This is the address of the instruction, plus 8. In other words, this base address is equal to the PC value used by the instruction.

2. Subtract the base address from the target address to form a byte offset. This offset is always a multiple of four, because all ARM instructions are word-aligned.

3. If the byte offset is outside the range −33554432 to +33554428, use an alternative code-generation strategy or produce an error as appropriate.

4. Otherwise, set the signed_immed_24 field of the instruction to bits{25:2} of the byte offset.

Notes

Memory bounds   Branching backwards past location zero and forwards over the end of the 32-bit address space is UNPREDICTABLE.

A4.1.6 BIC

31	28	27	26	25	24	23	22	21	20	19	16	15	12	11	0
cond		0	0	I	1	1	1	0	S	Rn		Rd		shifter_operand	
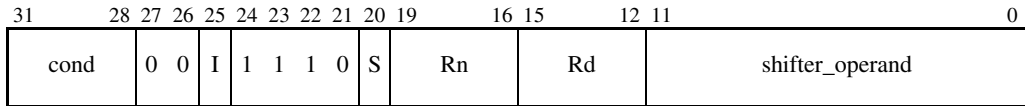
`BIC` (Bit Clear) performs a bitwise AND of one value with the complement of a second value. The first value comes from a register. The second value can be either an immediate value or a value from a register, and can be shifted before the BIC operation.

`BIC` can optionally update the condition code flags, based on the result.

Syntax

`BIC{<cond>}{S}  <Rd>, <Rn>, <shifter_operand>`

where:

`<cond>` Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-7. If `<cond>` is omitted, the `AL` (always) condition is used.

`S` Causes the S bit, bit[20], in the instruction to be set to 1 and specifies that the instruction updates the CPSR. If `S` is omitted, the S bit is set to 0 and the CPSR is not changed by the instruction. Two types of CPSR update can occur when `S` is specified:

- If `<Rd>` is not R15, the N and Z flags are set according to the result of the operation, and the C flag is set to the carry output bit generated by the shifter (see *Addressing Mode 1 - Data-processing operands* on page A5-2). The V flag and the rest of the CPSR are unaffected.

- If `<Rd>` is R15, the SPSR of the current mode is copied to the CPSR. This form of the instruction is UNPREDICTABLE if executed in User mode or System mode, because these modes do not have an SPSR.

`<Rd>` Specifies the destination register.

`<Rn>` Specifies the register that contains the first operand.

`<shifter_operand>`

Specifies the second operand. The options for this operand are described in *Addressing Mode 1 - Data-processing operands* on page A5-2, including how each option causes the I bit (bit[25]) and the shifter_operand bits (bits[11:0]) to be set in the instruction.

If the I bit is 0 and both bit[7] and bit[4] of shifter_operand are 1, the instruction is not `BIC`. Instead, see *Extending the instruction set* on page A3-35 to determine which instruction it is.

Architecture version

All.

Exceptions

None.

Operation

```
if ConditionPassed(cond) then
    Rd = Rn AND NOT shifter_operand
    if S == 1 and Rd == R15 then
        if CurrentModeHasSPSR() then
            CPSR = SPSR
        else UNPREDICTABLE
    else if S == 1 then
        N Flag = Rd[31]
        Z Flag = if Rd == 0 then 1 else 0
        C Flag = shifter_carry_out
        V Flag = unaffected
```

Usage

Use BIC to clear selected bits in a register. For each bit, BIC with 1 clears the bit, and BIC with 0 leaves it unchanged.

A4.1.7 BKPT
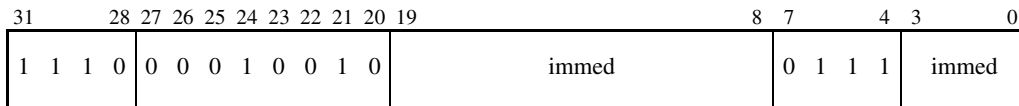
31			28	27	26	25	24	23	22	21	20	19		8	7		4	3		0
1	1	1	0	0	0	0	1	0	0	1	0	immed			0	1	1	immed		

BKPT (Breakpoint) causes a software breakpoint to occur. This breakpoint can be handled by an exception handler installed on the Prefetch Abort vector. In implementations that also include debug hardware, the hardware can optionally override this behavior and handle the breakpoint itself. When this occurs, the Prefetch Abort exception is taken.

Syntax

```
BKPT  <immediate>
```

where:

<immediate>   Is a 16-bit immediate value. The top 12 bits of <immediate> are placed in
                    bits[19:8] of the instruction, and the bottom 4 bits are placed in bits[3:0] of the
                    instruction. This value is ignored by the ARM hardware, but can be used by a
                    debugger to store additional information about the breakpoint.

Architecture version

Version 5 and above.

Exceptions

Prefetch Abort.

Operation

```
if (not overridden by debug hardware)
    R14_abt  = address of BKPT instruction + 4
    SPSR_abt = CPSR
    CPSR[4:0] = 0b10111            /* Enter Abort mode */
    CPSR[5]  = 0                   /* Execute in ARM state */
    /* CPSR[6] is unchanged */
    CPSR[7]  = 1                   /* Disable normal interrupts */
    if high vectors configured then
        PC   = 0xFFFF000C
    else
        PC   = 0x0000000C
```

Usage

The exact usage of BKPT depends on the debug system being used. A debug system can use the BKPT instruction in two ways:

*   Debug hardware, if present, does not override the normal behavior of the BKPT instruction, and so the Prefetch Abort vector is entered. If the system also allows real Prefetch Aborts to occur, the Prefetch Abort handler determines, in a system-dependent manner, whether the vector entry occurred as a result of a BKPT instruction or as a result of a real Prefetch Abort, and branches to debug code or Prefetch Abort code accordingly. Otherwise, the Prefetch Abort handler just branches straight to debug code.

    When used in this manner, the BKPT instruction must be avoided within abort handlers, as it corrupts R14_abt and SPSR_abt. For the same reason, it must also be avoided within FIQ handlers, since an FIQ interrupt can occur within an abort handler.

*   Debug hardware does override the normal behavior of the BKPT instruction and handles the software breakpoint itself. When finished, it typically either resumes execution at the instruction following the BKPT, or replaces the BKPT in memory with another instruction and resumes execution at that instruction.

    When BKPT is used in this manner, R14_abt and SPSR_abt are not corrupted, and so the above restrictions about its use in abort and FIQ handlers do not apply.

Notes

Condition field   BKPT is unconditional. If bits[31:28] of the instruction encode a valid condition other than the AL (always) condition, the instruction is UNPREDICTABLE.

Hardware override   Debug hardware in an implementation is specifically permitted to override the normal behavior of the BKPT instruction. Because of this, software must not use this instruction for purposes other than those documented by the debug system being used (if any). In particular, software cannot rely on the Prefetch Abort exception occurring, unless either there is guaranteed to be no debug hardware in the system or the debug system specifies that it occurs.

For more information, consult the documentation for the debug system being used.

A4.1.8   BLX (1)

31 30 29 28	27 26 25 24	23	0
1 1 1 1	1 0 1	H	signed_immed_24
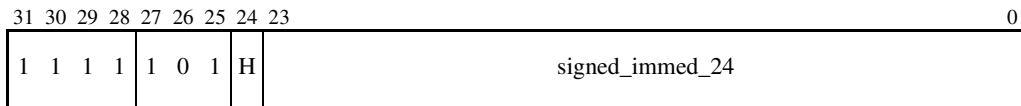
BLX (1) (Branch with Link and Exchange) calls a Thumb subroutine from the ARM instruction set at an address specified in the instruction.

This form of BLX is unconditional (always causing a change in program flow) and preserves the address of the instruction following the branch in the link register (R14). Execution of Thumb instructions begins at the target address.

Syntax

```
BLX  <target_addr>
```

where:

<target_addr>   Specifies the address of the Thumb instruction to branch to. The branch target address is calculated by:

1. Sign-extending the 24-bit signed (two's complement) immediate to 32 bits
2. Shifting the result left two bits
3. Setting bit[1] of the result of step 2 to the H bit
4. Adding the result of step 3 to the contents of the PC, which contains the address of the branch instruction plus 8.

The instruction can therefore specify a branch of approximately ±32MB (see *Usage* on page A4-17 for precise range).

Architecture version

Version 5 and above. See *The T bit* on page A2-13 for further details of operation on non-T variants.

Exceptions

None.

Operation

```
LR = address of the instruction after the BLX instruction
CPSR T bit = 1
PC = PC + (SignExtend(signed_immed_24) << 2) + (H << 1)
```

Usage

To return from a Thumb subroutine called via BLX to the ARM caller, use the Thumb instruction:

```
BX    R14
```

as described in *BX* on page A7-32, or use this instruction on subroutine entry:

```
PUSH {<registers>,R14}
```

and this instruction to return:

```
POP  {<registers>,PC}
```

To calculate the correct value of signed_immed_24, the assembler (or other toolkit component) must:

1.   Form the base address for this branch instruction. This is the address of the instruction, plus 8. In other words, this base address is equal to the PC value used by the instruction.

2.   Subtract the base address from the target address to form a byte offset. This offset is always even, because all ARM instructions are word-aligned and all Thumb instructions are halfword-aligned.

3.   If the byte offset is outside the range −33554432 to +33554430, use an alternative code-generation strategy or produce an error as appropriate.

4.   Otherwise, set the signed_immed_24 field of the instruction to bits[25:2] of the byte offset, and the H bit of the instruction to bit[1] of the byte offset.

Notes

Condition   Unlike most other ARM instructions, this instruction cannot be executed conditionally.

Bit[24]   This bit is used as bit[1] of the target address.

A4.1.9   BLX (2)

31 30 29 28	27 26 25 24 23 22 21 20	19 16	15 12	11 8	7 6 5 4	3 0
cond	0 0 0 1 0 0 1 0	SBO	SBO	SBO	0 0 1 1	Rm

`BLX` (2) calls an ARM or Thumb subroutine from the ARM instruction set, at an address specified in a register.

It sets the CPSR T bit to bit[0] of Rm. This selects the instruction set to be used in the subroutine.

The branch target address is the value of register Rm, with its bit[0] forced to zero.

It sets R14 to a return address. To return from the subroutine, use a `BX R14` instruction, or store R14 on the stack and reload the stored value into the PC.

Syntax

```
BLX{<cond>}   <Rm>
```

where:

<cond>   Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-7. If `<cond>` is omitted, the `AL` (always) condition is used.

<Rm>   Is the register containing the address of the target instruction. Bit[0] of Rm is 0 to select a target ARM instruction, or 1 to select a target Thumb instruction. If R15 is specified for `<Rm>`, the results are UNPREDICTABLE.

Architecture version

Version 5 and above. See *The T bit* on page A2-13 for further details of operation on non-T variants.

Exceptions

None.

Operation

```
if ConditionPassed(cond) then
    LR = address of instruction after the BLX instruction
    CPSR T bit = Rm[0]
    PC = Rm AND 0xFFFFFFFE
```

Notes

ARM/Thumb state transfers

   If Rm[1:0] == 0b10, the result is UNPREDICTABLE, as branches to non word-aligned addresses are impossible in ARM state.

ARM DDI 0100F-02

A4.1.10 BX

31	28	27 26 25 24 23 22 21 20	19	16	15	12	11	8	7 6 5 4	3	0
cond		0 0 0 1 0 0 1 0	SBO		SBO		SBO		0 0 0 1	Rm	
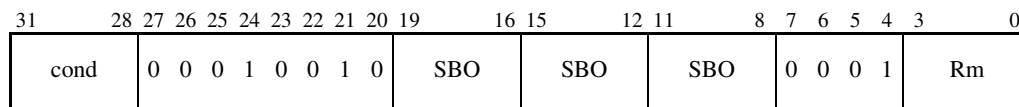
BX (Branch and Exchange) branches to an address, with an optional switch to Thumb state.

Syntax

```
BX{<cond>}   <Rm>
```

where:

<cond>   Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-7. If <cond> is omitted, the AL (always) condition is used.

<Rm>   Holds the value of the branch target address. Bit[0] of Rm is 0 to select a target ARM instruction, or 1 to select a target Thumb instruction.

Architecture version

Version 5 and above, and T variants of version 4. See *The T bit* on page A2-13 for further details of operation on non-T variants of version 5.

Exceptions

None.

Operation

```
if ConditionPassed(cond) then
    CPSR T bit = Rm[0]
    PC = Rm AND 0xFFFFFFFE
```

Notes

ARM/Thumb state transfers

If Rm[1:0] == 0b10, the result is UNPREDICTABLE, as branches to non word-aligned addresses are impossible in ARM state.

Use of R15   Register 15 can be specified for <Rm>, but doing so is discouraged.

In a BX R15 instruction, R15 is read as normal for ARM code, that is, it is the address of the BX instruction itself plus 8. The result is to branch to the second following word, executing in ARM state. This is precisely the same effect that would have been obtained if a B instruction with an offset field of 0 had been executed, or an ADD PC, PC, #0 or MOV PC, PC instruction. In new code, use these instructions in preference to the more complex BX PC instruction.

A4.1.11 BXJ

31	28	27 26 25 24 23 22 21 20	19 16	15 12	11 8	7 6 5 4	3 0
cond		0 0 0 1 0 0 1 0	SBO	SBO	SBO	0 0 1 0	Rm
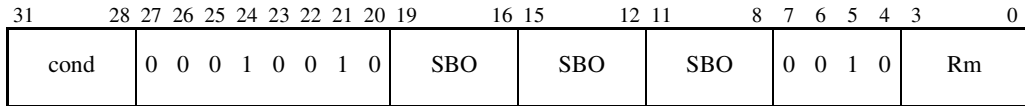
BXJ (Branch and change to Java state) enters Java state if Java is available and enabled. Otherwise BXJ behaves exactly as BX (see *BX* on page A4-19).

Syntax

```
BXJ{<cond>}   <Rm>
```

where:

<cond>   Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-7. If <cond> is omitted, the AL (always) condition is used.

<Rm>   Holds the value of the branch target address for use if Java state is not available. Bit[0] of Rm is 0 to select a target ARM instruction, or 1 to select a target Thumb instruction.

Architecture version

Version 6 and above, plus J variants of version 5.

Exceptions

None.

Operation

```
if ConditionPassed(cond) then
    if (JE bit of Main Configuration register) == 0 then
        CPSR T bit = Rm[0]
        PC = Rm AND 0xFFFFFFFE
    else
        jpc = SUBARCHITECTURE DEFINED value
        if (CV bit of Java OS Control register) == 0 then
            PC = invalidhandler
        else
            J Flag = 1
            Start bytecode execution at jpc
    else
        if ((CV bit of Javao OS Control register) == 0) AND
                        (IMPLEMENTATION DEFINED CONDITION) then
            PC = invalidhandler
        else
            /* Subject to SUBARCHITECTURE DEFINED restrictions on Rm: */
            CPSR T bit = Rm[0]
            PC = Rm AND 0xFFFFFFFE
```

Usage

Do not use BXJ unless one of the following conditions is true:

* The JE bit of the Main Configuration Register is 0.

* The Enabled Java Virtual Machine in use conforms to all the subarchitecture defined restrictions of the Java Extension hardware being used.

Notes

ARM/Thumb state transfers

If Rm[1:0] == 0b10, the result is UNPREDICTABLE, as branches to non word-aligned addresses are impossible in ARM state.

Use of R15   If register 15 is specified for <Rm>, the result is UNPREDICTABLE.

Java bytecode address

The Java bytecode address is determined in a subarchitecture defined manner, typically from the contents of a specific general-purpose register, the *Java Program Counter* (jpc).

A4.1.12 CDP

31	28	27	26	25	24	23	20	19	16	15	12	11	8	7	5	4	3	0
cond		1	1	1	0	opcode_1		CRn		CRd		cp_num		opcode_2		0	CRm	
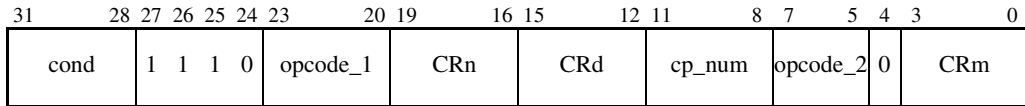
CDP (Coprocessor Data Processing) tells the coprocessor whose number is cp_num to perform an operation that is independent of ARM registers and memory. If no coprocessors indicate that they can execute the instruction, an Undefined Instruction exception is generated.

Syntax

```
CDP{<cond>}  <coproc>, <opcode_1>, <CRd>, <CRn>, <CRm>, <opcode_2>
CDP2         <coproc>, <opcode_1>, <CRd>, <CRn>, <CRm>, <opcode_2>
```

where:

<cond>	Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-7. If <cond> is omitted, the AL (always) condition is used.
CDP2	Causes the condition field of the instruction to be set to 0b1111. This provides additional opcode space for coprocessor designers. The resulting instructions can only be executed unconditionally.
<coproc>	Specifies the name of the coprocessor, and causes the corresponding coprocessor number to be placed in the cp_num field of the instruction. The standard generic coprocessor names are p0, p1, ..., p15.
<opcode_1>	Specifies (in a coprocessor-specific manner) which coprocessor operation is to be performed.
<CRd>	Specifies the destination coprocessor register for the instruction.
<CRn>	Specifies the coprocessor register that contains the first operand.
<CRm>	Specifies the coprocessor register that contains the second operand.
<opcode_2>	Specifies (in a coprocessor-specific manner) which coprocessor operation is to be performed.

Architecture version

CDP is in Version 2 and above.

CDP2 is in Version 5 and above.

Exceptions

Undefined Instruction.

Operation

```
if ConditionPassed(cond) then
    Coprocessor[cp_num]-dependent operation
```

Usage

Use CDP to initiate coprocessor instructions that do not operate on values in ARM registers or in main memory. An example is a floating-point multiply instruction for a floating-point coprocessor.

Notes

Coprocessor fields Only instruction bits[31:24], bits[11:8], and bit[4] are architecturally defined. The remaining fields are recommendations, for compatibility with ARM Development Systems.

Unimplemented coprocessor instructions

 Hardware coprocessor support is optional, regardless of the architecture version. An implementation can choose to implement a subset of the coprocessor instructions, or no coprocessor instructions at all. Any coprocessor instructions that are not implemented instead cause an undefined instruction exception.

A4.1.13 CLZ

31	28	27 26 25 24 23 22 21 20	19	16	15	12	11	8	7 6 5 4	3	0
cond		0 0 0 1 0 1 1 0	SBO		Rd		SBO		0 0 0 1	Rm	
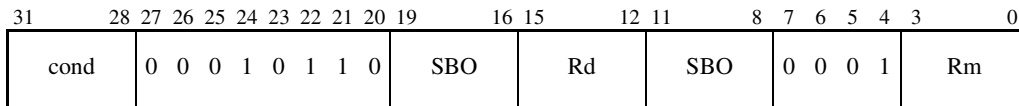
CLZ (Count Leading Zeros) returns the number of binary zero bits before the first binary one bit in a value.

CLZ does not update the condition code flags.

Syntax

```
CLZ{<cond>}   <Rd>, <Rm>
```

where:

<cond>   Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-7. If <cond> is omitted, the AL (always) condition is used.

<Rd>   Specifies the destination register for the operation. If R15 is specified for <Rd>, the result is UNPREDICTABLE.

<Rm>   Specifies the source register for this operation. If R15 is specified for <Rm>, the result is UNPREDICTABLE.

Architecture version

Version 5 and above.

Exceptions

None.

Operation

```
if Rm == 0
    Rd = 32
else
    Rd = 31 - (bit position of most significant '1' in Rm)
```

Usage

Use CLZ followed by a left shift of Rm by the resulting Rd value to normalize the value of register Rm. This shifts Rm so that its most significant 1 bit is in bit[31]. Using MOVS rather than MOV sets the Z flag in the special case that Rm is zero and so does not have a most significant 1 bit:

```
    CLZ   Rd, Rm
    MOVS  Rm, Rm, LSL Rd
```

ARM DDI 0100F-02

A4.1.14 CMN

31	28	27	26	25	24	23	22	21	20	19	16	15	12	11	0
cond		0	0	I	1	0	1	1	1	Rn		SBZ		shifter_operand	
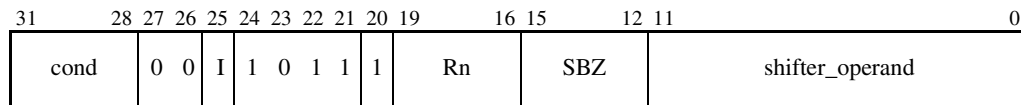
CMN (Compare Negative) compares one value with the two's complement of a second value. The first value comes from a register. The second value can be either an immediate value or a value from a register, and can be shifted before the comparison.

CMN updates the condition flags, based on the result of adding the two values.

Syntax

```
CMN{<cond>}  <Rn>, <shifter_operand>
```

where:

<cond>   Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-7. If <cond> is omitted, the AL (always) condition is used.

<Rn>   Specifies the register that contains the first operand.

<shifter_operand>

               Specifies the second operand. The options for this operand are described in *Addressing Mode 1 - Data-processing operands* on page A5-2, including how each option causes the I bit (bit[25]) and the shifter_operand bits (bits[11:0]) to be set in the instruction.

               If the I bit is 0 and both bit[7] and bit[4] of shifter_operand are 1, the instruction is not CMN. Instead, see *Extending the instruction set* on page A3-35 to determine which instruction it is.

Architecture version

All.

Exceptions

None.

Operation

```
if ConditionPassed(cond) then
    alu_out = Rn + shifter_operand
    N Flag = alu_out[31]
    Z Flag = if alu_out == 0 then 1 else 0
    C Flag = CarryFrom(Rn + shifter_operand)
    V Flag = OverflowFrom(Rn + shifter_operand)
```

Usage

WRITE A sensible usage section. TBD

CMN performs a comparison by adding the value of `<shifter_operand>` to the value of register `<Rn>`, and updates the condition code flags (based on the result). This is almost equivalent to subtracting the negative of the second operand from the first operand, and setting the flags on the result.

The difference is that the flag values generated can differ when the second operand is 0 or `0x80000000`. For example, this instruction always leaves the C flag = 1:

```
CMP Rn, #0
```

and this instruction always leaves the C flag = 0:

```
CMN Rn, #0
```

A4.1.15 CMP

31	28	27	26	25	24	23	22	21	20	19	16	15	12	11	0
cond		0	0	I	1	0	1	0	1	Rn		SBZ		shifter_operand	
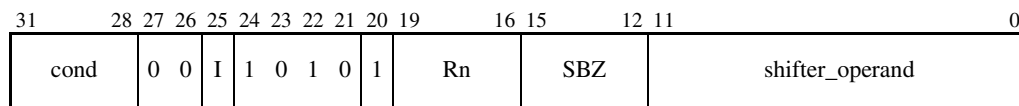
CMP (Compare) compares two values. The first value comes from a register. The second value can be either an immediate value or a value from a register, and can be shifted before the comparison.

CMP updates the condition flags, based on the result of subtracting the second value from the first.

Syntax

```
CMP{<cond>}  <Rn>, <shifter_operand>
```

where:

<cond>   Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-7. If <cond> is omitted, the AL (always) condition is used.

<Rn>   Specifies the register that contains the first operand.

<shifter_operand>

        Specifies the second operand. The options for this operand are described in *Addressing Mode 1 - Data-processing operands* on page A5-2, including how each option causes the I bit (bit[25]) and the shifter_operand bits (bits[11:0]) to be set in the instruction.

        If the I bit is 0 and both bit[7] and bit[4] of shifter_operand are 1, the instruction is not CMP. Instead, see *Extending the instruction set* on page A3-35 to determine which instruction it is.

Architecture version

All.

Exceptions

None.

Operation

```
if ConditionPassed(cond) then
    alu_out = Rn - shifter_operand
    N Flag = alu_out[31]
    Z Flag = if alu_out == 0 then 1 else 0
    C Flag = NOT BorrowFrom(Rn - shifter_operand)
    V Flag = OverflowFrom(Rn - shifter_operand)
```

A4.1.16 CPS

31 30 29 28	27 26 25 24 23 22 21 20	19 18	17	16	15 9	8	7	6	5	4 0
1 1 1 1	0 0 0 1 0 0 0 0	imod	mmod	0	SBZ	A	I	F	0	mode
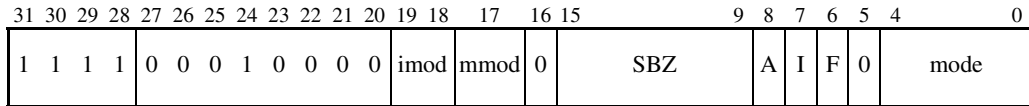
CPS (Change Processor State) changes one or more of the mode, A, I, and F bits of the CPSR, without changing the other CPSR bits.

Syntax

```
CPS<effect> <iflags> {, #<mode>}

CPS     #<mode>
```

where:

<effect>   Specifies what effect is wanted on the interrupt disable bits A, I, and F in the CPSR. This is one of:

   IE   Interrupt Enable, encoded by imod == 0b10. This sets the specified bits to 0.

   ID   Interrupt Disable, encoded by imod == 0b11. This sets the specified bits to 1.

   If <effect> is specified, the bits to be affected are specified by <iflags>. These are encoded in the A, I, and F bits of the instruction. The mode can optionally be changed by specifying a mode number as <mode>.

   If <effect> is not specified, then:
   • <iflags> is not specified and interrupt settings are not changed
   • the A, I, and F bits of the instruction are zero
   • imod = 0b00
   • <mode> specifies the new mode number.

<iflags>   Is a sequence of one or more of the following, specifying which interrupt disable flags are affected:

   a   Sets the A bit in the instruction, causing the specified effect on the CPSR A (imprecise data abort) bit.

   i   Sets the I bit in the instruction, causing the specified effect on the CPSR I (IRQ interrupt) bit.

   f   Sets the F bit in the instruction, causing the specified effect on the CPSR F (FIQ interrupt) bit.

<mode>   Specifies the number of the mode to change to. If it is present, then mmod == 1 and the mode number is encoded in the mode field of the instruction. If it is omitted, then mmod == 0 and the mode field of the instruction is zero. See *Mode bits* on page A2-14 for details.

ARM DDI 0100F-02

Architecture version

Version 6 and above.

Exceptions

None.

Operation

```
if InAPrivilegedMode() then
    if imod[1] == 1 then
        if A == 1 then CPSR[8] = imod[0]
        if I == 1 then CPSR[7] = imod[0]
        if F == 1 then CPSR[6] = imod[0]
    /* else no change to interrupt disable bits */
    if mmod == 1 then
        CPSR[4:0] = mode
```

Notes

User mode   `CPS` has no effect in User mode.

Meaningless bit combinations

The following combinations of imod and mmod are meaningless:

- imod == 0b00, mmod == 0
- imod == 0b01, mmod == 0
- imod == 0b01, mmod == 1

An assembler must not generate them. The effects are UNPREDICTABLE on execution.

Condition   Unlike most other ARM instructions, `CPS` cannot be executed conditionally.

Examples

```
CPSIE   a,#31   ; enable imprecise data aborts, change to System mode
CPSID   if      ; disable interrupts and fast interrupts
CPS     #16     ; change to User mode
```

A4.1.17 CPY

31	28	27 26 25 24 23 22 21 20	19 16	15 12	11 10 9 8 7 6 5 4	3 0
cond		0 0 0 1 1 0 1 0	SBZ	Rd	0 0 0 0 0 0 0 0	Rm

CPY (Copy) copies a value from one register to another. It is a synonym for MOV, with no flag setting and no shift. See *MOV* on page A4-67.

Syntax

```
CPY{<cond>}   <Rd>, <Rm>
```

where:

<cond>   Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-7. If <cond> is omitted, the AL (always) condition is used.

<Rd>   Specifies the destination register.

<Rm>   Specifies the source register.
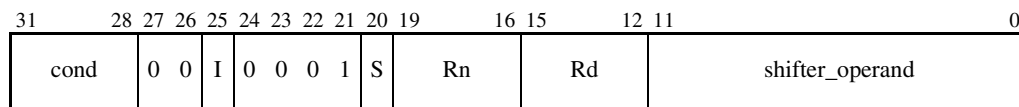
Architecture version

Version 6 and above.

Exceptions

None.

Operation

```
if ConditionPassed(cond) then
    Rd = Rm
```

       ARM DDI 0100F-02
Confidential - Draft

A4.1.18 EOR

31	28	27	26	25	24	23	22	21	20	19	16	15	12	11	0
cond		0	0	I	0	0	0	1	S	Rn		Rd		shifter_operand	

`EOR` (Exclusive OR) performs a bitwise Exclusive-OR of two values. The first value comes from a register. The second value can be either an immediate value or a value from a register, and can be shifted before the exclusive OR operation.

`EOR` can optionally update the condition code flags, based on the result.

Syntax

`EOR{<cond>}{S}  <Rd>, <Rn>, <shifter_operand>`

where:

<cond>   Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-7. If `<cond>` is omitted, the `AL` (always) condition is used.

S   Sets the S bit (bit[20]) in the instruction to 1 and specifies that the instruction updates the CPSR. If `S` is omitted, the S bit is set to 0 and the CPSR is not changed by the instruction. Two types of CPSR update can occur when `S` is specified:

   - If `<Rd>` is not R15, the N and Z flags are set according to the result of the operation, and the C flag is set to the carry output bit generated by the shifter (see *Addressing Mode 1 - Data-processing operands* on page A5-2). The V flag and the rest of the CPSR are unaffected.

   - If `<Rd>` is R15, the SPSR of the current mode is copied to the CPSR. This form of the instruction is UNPREDICTABLE if executed in User mode or System mode, because these modes do not have an SPSR.

<Rd>   Specifies the destination register.

<Rn>   Specifies the register that contains the first operand.

<shifter_operand>

   Specifies the second operand. The options for this operand are described in *Addressing Mode 1 - Data-processing operands* on page A5-2, including how each option causes the I bit (bit[25]) and the shifter_operand bits (bits[11:0]) to be set in the instruction.

   If the I bit is 0 and both bit[7] and bit[4] of shifter_operand are 1, the instruction is not `EOR`. Instead, see *Extending the instruction set* on page A3-35 to determine which instruction it is.

Architecture version

All.

Confidential - Draft

Exceptions

None.

Operation

```
if ConditionPassed(cond) then
    Rd = Rn EOR shifter_operand
    if S == 1 and Rd == R15 then
        if CurrentModeHasSPSR() then
            CPSR = SPSR
        else UNPREDICTABLE
    else if S == 1 then
        N Flag = Rd[31]
        Z Flag = if Rd == 0 then 1 else 0
        C Flag = shifter_carry_out
        V Flag = unaffected
```

Usage

Use EOR to invert selected bits in a register. For each bit, EOR with 1 inverts that bit, and EOR with 0 leaves it unchanged.

A4.1.19 LDC

31	28	27	26	25	24	23	22	21	20	19	16	15	12	11	8	7	0
cond		1	1	0	P	U	N	W	1	Rn		CRd		cp_num		8_bit_word_offset	
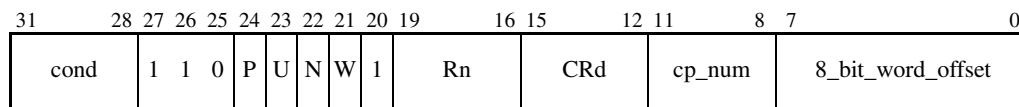
`LDC` (Load Coprocessor) loads memory data from a sequence of consecutive memory addresses to a coprocessor.

If no coprocessors indicate that they can execute the instruction, an Undefined Instruction exception is generated.

Syntax

```
LDC{<cond>}{L}  <coproc>, <CRd>, <addressing_mode>
LDC2{L}         <coproc>, <CRd>, <addressing_mode>
```

where:

<cond>   Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-7. If `<cond>` is omitted, the `AL` (always) condition is used.

LDC2   Causes the condition field of the instruction to be set to 0b1111. This provides additional opcode space for coprocessor designers. The resulting instructions can only be executed unconditionally.

L   Sets the N bit (bit[22]) in the instruction to 1 and specifies a long load (for example, double-precision instead of single-precision data transfer). If L is omitted, the N bit is 0 and the instruction specifies a short load.

<coproc>   Specifies the name of the coprocessor, and causes the corresponding coprocessor number to be placed in the cp_num field of the instruction. The standard generic coprocessor names are p0, p1, ..., p15.

<CRd>   Specifies the coprocessor destination register.

<addressing_mode>

                Is described in *Addressing Mode 5 - Load and Store Coprocessor* on page A5-60. It determines the P, U, Rn, W and 8_bit_word_offset bits of the instruction.

                The syntax of all forms of `<addressing_mode>` includes a *base register* `<Rn>`. Some forms also specify that the instruction modifies the base register value (this is known as *base register write-back*).

Architecture version

`LDC` is in Version 2 and above.

`LDC2` is in Version 5 and above.

Exceptions

Undefined Instruction, Data Abort.

Operation

```
if ConditionPassed(cond) then
    address = start_address
    load Memory[address,4] for Coprocessor[cp_num]
    while (NotFinished(Coprocessor[cp_num]))
        address = address + 4
        load Memory[address,4] for Coprocessor[cp_num]
    assert address == end_address
```

Usage

LDC is useful for loading coprocessor data from memory.

Notes

Coprocessor fields Only instruction bits[31:23], bits[21:16], and bits[11:0] are ARM
architecture-defined. The remaining fields (bit[22] and bits[15:12]) are
recommendations, for compatibility with ARM Development Systems.

In the case of the Unindexed addressing mode (P==0, U==1, W==0), instruction
bits[7:0] are also not defined by the ARM architecture, and can be used to specify
additional coprocessor options.

Data Abort For details of the effects of the instruction if a Data Abort occurs, see *Effects of
data-aborted instructions* on page A2-21.

Non word-aligned addresses

Load coprocessor register instructions ignore the least significant two bits of
address.

Alignment If an implementation includes a System Control coprocessor (see Chapter B3 *The
System Control Coprocessor*), and alignment checking is enabled, an address with
bits[1:0] != 0b00 causes an alignment exception.

Unimplemented coprocessor instructions

Hardware coprocessor support is optional, regardless of the architecture version.
An implementation can choose to implement a subset of the coprocessor
instructions, or no coprocessor instructions at all. Any coprocessor instructions that
are not implemented instead cause an undefined instruction exception.

                   ARM DDI 0100F-02

A4.1.20 LDM (1)

31	28	27 26 25	24	23	22	21	20	19	16	15	0
cond		1 0 0	P	U	0	W	1	Rn		register_list	
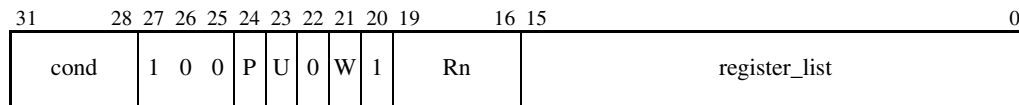
LDM (1) (Load Multiple) loads a non-empty subset, or possibly all, of the general-purpose registers from sequential memory locations. It is useful for block loads, stack operations and procedure exit sequences.

The general-purpose registers loaded can include the PC. If they do, the word loaded for the PC is treated as an address and a branch occurs to that address. In ARM architecture version 5 and above, bit[0] of the loaded value determines whether execution continues after this branch in ARM state or in Thumb state, as though a BX (loaded_value) instruction had been executed (but see also *The T bit* on page A2-13 for operation on non-T variants of ARM architecture version 5). In earlier versions of the architecture, bits[1:0] of the loaded value are ignored and execution continues in ARM state, as though the instruction MOV PC,(loaded_value) had been executed.

Syntax

LDM{<cond>}<addressing_mode> <Rn>{!}, <registers>

where:

<cond>   Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-7. If <cond> is omitted, the AL (always) condition is used.

<addressing_mode>

Is described in *Addressing Mode 4 - Load and Store Multiple* on page A5-52. It determines the P, U, and W bits of the instruction.

<Rn>   Specifies the base register used by <addressing_mode>. Using R15 as the base register <Rn> gives an UNPREDICTABLE result.

!   Sets the W bit, causing the instruction to write a modified value back to its base register Rn as specified in *Addressing Mode 4 - Load and Store Multiple* on page A5-52. If ! is omitted, the W bit is 0 and the instruction does not change its base register in this way. (However, if the base register is included in <registers>, it changes when a value is loaded into it.)

<registers>

Is a list of registers, separated by commas and surrounded by { and }. It specifies the set of registers to be loaded by the LDM instruction.

The registers are loaded in sequence, the lowest-numbered register from the lowest memory address (start_address), through to the highest-numbered register from the highest memory address (end_address). If the PC is specified in the register list (opcode bit[15] is set), the instruction causes a branch to the address (data) loaded into the PC.

For each of i=0 to 15, bit[i] in the register_list field of the instruction is 1 if Ri is in the list and 0 otherwise. If bits[15:0] are all zero, the result is UNPREDICTABLE.

Copyright © 1996–2003 ARM Limited. All rights reserved.
Confidential - Draft

Architecture version

All.

Exceptions

Data Abort.

Operation

```
if ConditionPassed(cond) then
    address = start_address

    for i = 0 to 14
        if register_list[i] == 1 then
            Ri = Memory[address,4]
            address = address + 4

    if register_list[15] == 1 then
        value = Memory[address,4]
        if (architecture version 5 or above) then
            pc = value AND 0xFFFFFFFE
            T Bit = value[0]
        else
            pc = value AND 0xFFFFFFFC
        address = address + 4

    assert end_address = address - 4
```

Notes

Operand restrictions

If the base register <Rn> is specified in <registers>, and base register write-back is specified, the final value of <Rn> is UNPREDICTABLE.

Data Abort   For details of the effects of the instruction if a Data Abort occurs, see *Effects of data-aborted instructions* on page A2-21.

Non word-aligned addresses

LDM (1) ignores the least significant two bits of address.

Alignment   In ARM architecture v6, and earlier implementations if they include a System Control coprocessor (see Chapter B3 *The System Control Coprocessor*), an address with bits[1:0] != 0b00 causes an alignment exception if alignment checking is enabled.

ARM/Thumb state transfers (ARM architecture version 5 and above)

If bits[1:0] of a value loaded for R15 are 0b10, the result is UNPREDICTABLE, as branches to non word-aligned addresses are impossible in ARM state.

Time order   The time order of the accesses to individual words of memory generated by this instruction is only defined in some circumstances. See *Data accesses to memory-mapped I/O* on page B2-27 for details.

A4.1.21 LDM (2)

31	28	27	26	25	24	23	22	21	20	19	16	15	14	0
cond		1	0	0	P	U	1	0	1	Rn		0	register_list	

LDM (2) loads User mode registers when the processor is in a privileged mode. This is useful when performing process swaps, and in instruction emulators. LDM (2) loads a non-empty subset of the User mode general-purpose registers from sequential memory locations.

Syntax

```
LDM{<cond>}<addressing_mode>  <Rn>, <registers_without_pc>^
```

where:

<cond>   Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-7. If <cond> is omitted, the AL (always) condition is used.

<addressing_mode>

Is described in *Addressing Mode 4 - Load and Store Multiple* on page A5-52. It determines the P and U bits of the instruction. Only the forms of this addressing mode with W == 0 are available for this form of the LDM instruction.

<Rn>   Specifies the base register used by <addressing_mode>. Using R15 as <Rn> gives an UNPREDICTABLE result.

<registers_without_pc>

Is a list of registers, separated by commas and surrounded by { and }. This list must not include the PC, and specifies the set of registers to be loaded by the LDM instruction.

The registers are loaded in sequence, the lowest-numbered register from the lowest memory address (start_address), through to the highest-numbered register from the highest memory address (end_address).

For each of i=0 to 15, bit[i] in the register_list field of the instruction is 1 if Ri is in the list and 0 otherwise. If bits[15:0] are all zero, the result is UNPREDICTABLE.

^   For an LDM instruction that does not load the PC, this indicates that User mode registers are to be loaded.

Architecture version

All.

Exceptions

Data Abort.

Operation

```
if ConditionPassed(cond) then
    address = start_address
    for i = 0 to 14
        if register_list[i] == 1
            Ri_usr = Memory[address,4]
            address = address + 4
    assert end_address == address - 4
```

Notes

Write-back Setting bit[21] (the W bit) has UNPREDICTABLE results.

User and System mode

This form of LDM is UNPREDICTABLE in User mode or System mode.

Base register mode The base register is read from the current processor mode registers, not the User mode registers.

Data Abort For details of the effects of the instruction if a Data Abort occurs, see *Effects of data-aborted instructions* on page A2-21.

Non word-aligned addresses

LDM instructions ignore the least significant two bits of address.

Alignment In ARM architecture v6, and earlier implementations if they include a System Control coprocessor (see Chapter B3 *The System Control Coprocessor*), an address with bits[1:0] != 0b00 causes an alignment exception if alignment checking is enabled.

Time order The time order of the accesses to individual words of memory generated by this instruction is only defined in some circumstances. See *Data accesses to memory-mapped I/O* on page B2-27 for details.

Banked registers In ARM architecture v5 and below, LDM (2) must not be followed by an instruction that accesses banked registers. A following NOP is a good way to ensure this.

A4.1.22 LDM (3)

31	28 27 26 25	24	23	22	21	20	19 16	15	14 0
cond	1 0 0	P	U	1	W	1	Rn	1	register_list
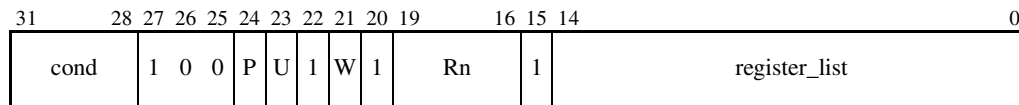
LDM (3) loads a subset, or possibly all, of the general-purpose registers and the PC from sequential memory locations. Also, the SPSR of the current mode is copied to the CPSR. This is useful for returning from an exception.

The value loaded for the PC is treated as an address and a branch occurs to that address. In ARM architecture version 5 and above, and in T variants of version 4, the value copied from the SPSR T bit to the CPSR T bit determines whether execution continues after the branch in ARM state or in Thumb state (but see also *The T bit* on page A2-13 for operation on non-T variants of ARM architecture version 5). In earlier architecture versions, it continues after the branch in ARM state (the only possibility in those architecture versions).

Syntax

```
LDM{<cond>}<addressing_mode>  <Rn>{!}, <registers_and_pc>^
```

where:

<cond>   Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-7. If <cond> is omitted, the AL (always) condition is used.

<addressing_mode>
                    Is described in *Addressing Mode 4 - Load and Store Multiple* on page A5-52. It determines the P, U, and W bits of the instruction.

<Rn>   Specifies the base register used by <addressing_mode>. Using R15 as <Rn> gives an UNPREDICTABLE result.

!   Sets the W bit, and the instruction writes a modified value back to its base register Rn (see *Addressing Mode 4 - Load and Store Multiple* on page A5-52). If ! is omitted, the W bit is 0 and the instruction does not change its base register in this way. (However, if the base register is included in <registers>, it changes when a value is loaded into it.)

<registers_and_pc>
                    Is a list of registers, separated by commas and surrounded by { and }. This list must include the PC, and specifies the set of registers to be loaded by the LDM instruction.

                    The registers are loaded in sequence, the lowest-numbered register from the lowest memory address (start_address), through to the highest-numbered register from the highest memory address (end_address).

                    For each of i=0 to 15, bit[i] in the register_list field of the instruction is 1 if Ri is in the list and 0 otherwise.

^   For an LDM instruction that loads the PC, this indicates that the SPSR of the current mode is copied to the CPSR.

Architecture version

All.

Exceptions

Data Abort.

Operation

```
if ConditionPassed(cond) then
    address = start_address

    for i = 0 to 14
        if register_list[i] == 1 then
            Ri = Memory[address,4]
            address = address + 4

    if CurrentModeHasSPSR() then
        CPSR = SPSR
    else UNPREDICTABLE

    value = Memory[address,4]
    if (architecture version 4T, 5 or above) and (T Bit == 1) then
        PC = value AND 0xFFFFFFFE
    else
        PC = value AND 0xFFFFFFFC
    address = address + 4

    assert end_address = address - 4
```

Notes

User and System mode

This instruction is UNPREDICTABLE in User or System mode.

Operand restrictions

If the base register `<Rn>` is specified in `<registers_and_pc>`, and base register write-back is specified, the final value of `<Rn>` is UNPREDICTABLE.

Data Abort   For details of the effects of the instruction if a Data Abort occurs, see *Effects of data-aborted instructions* on page A2-21.

Non word-aligned addresses

Load Multiple instructions ignore the least significant two bits of address.

Alignment   In ARM architecture v6, and earlier implementations if they include a System Control coprocessor (see Chapter B3 *The System Control Coprocessor*), an address with bits[1:0] != 0b00 causes an alignment exception if alignment checking is enabled.

ARM/Thumb state transfers (ARM architecture versions 4T, 5 and above)

If the SPSR T bit is 0 and bit[1] of the value loaded into the PC is 1, the results are UNPREDICTABLE because it is not possible to branch to an ARM instruction at a non word-aligned address. Note that no special precautions against this are needed on normal exception returns, because exception entries always either set the T bit of the SPSR to 1 or bit[1] of the return link value in R14 to 0.

Time order   The time order of the accesses to individual words of memory generated by this instruction is not defined. See *Data accesses to memory-mapped I/O* on page B2-27 for details.

A4.1.23 LDR

31	28	27	26	25	24	23	22	21	20	19	16	15	12	11	0
cond		0	1	I	P	U	0	W	1	Rn		Rd		addr_mode	
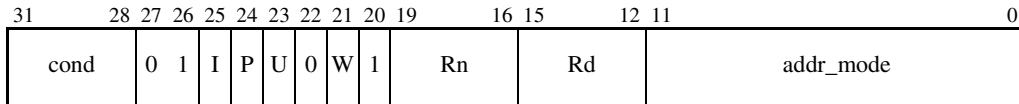
LDR (Load Register) loads a word from a memory address.

If the PC is specified as register <Rd>, the instruction loads a data word which it treats as an address, then branches to that address. In ARM architecture version 5 and above, bit[0] of the loaded value determines whether execution continues after this branch in ARM state or in Thumb state, as though a BX (loaded_value) instruction had been executed (but see also *The T bit* on page A2-13 for operation on non-T variants of ARM architecture version 5). In earlier versions of the architecture, bits[1:0] of the loaded value are ignored and execution continues in ARM state, as though a MOV PC, (loaded_value) instruction had been executed.

Syntax

LDR{<cond>} <Rd>, <addressing_mode>

where:

<cond>   Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-7. If <cond> is omitted, the AL (always) condition is used.

<Rd>   Specifies the destination register for the loaded value.

<addressing_mode>

             Is described in *Addressing Mode 2 - Load and Store Word or Unsigned Byte* on page A5-19. It determines the I, P, U, W, Rn and addr_mode bits of the instruction.

             The syntax of all forms of <addressing_mode> includes a *base register* <Rn>. Some forms also specify that the instruction modifies the base register value (this is known as *base register write-back*).

Architecture version

All.

Exceptions

Data Abort.

ARM DDI 0100F-02

Operation

```
if ConditionPassed(cond) then
    if address[1:0] == 0b00 then
        value = Memory[address,4]
    else if address[1:0] == 0b01 then
        value = Memory[address,4] Rotate_Right 8
    else if address[1:0] == 0b10 then
        value = Memory[address,4] Rotate_Right 16
    else /* address[1:0] == 0b11 */
        value = Memory[address,4] Rotate_Right 24

    if (Rd is R15) then
        if (architecture version 5 or above) then
            PC = value AND 0xFFFFFFFE
            T Bit = value[0]
        else
            PC = value AND 0xFFFFFFFC
    else
        Rd = value
```

Usage

Using the PC as the base register allows PC-relative addressing, which facilitates position-independent code. Combined with a suitable addressing mode, LDR allows 32-bit memory data to be loaded into a general-purpose register where its value can be manipulated. If the destination register is the PC, this instruction loads a 32-bit address from memory and branches to that address.

To synthesize a Branch with Link, precede the LDR instruction with MOV LR, PC.

Alignment

ARM architecture version 5 and below

> If the address is not word-aligned, the loaded value is rotated right by 8 times the value of bits[1:0] of the address. For a little-endian memory system, this rotation causes the addressed byte to occupy the least significant byte of the register. For a big-endian memory system, it causes the addressed byte to occupy bits[31:24] or bits[15:8] of the register, depending on whether bit[0] of the address is 0 or 1 respectively.

> If an implementation includes a System Control coprocessor (see Chapter B3 *The System Control Coprocessor*), and alignment checking is enabled, an address with bits[1:0] != 0b00 causes an alignment exception.

ARM architecture version 6 and above

> You can select any one of the following three behaviors:

> • The same behavior as in ARM architecture version 5.

> • True unaligned access. Bits[1:0] of the address specify the location in memory of the first of four consecutive bytes to load.

•   Strict alignment checking. Bits[1:0] of the address must be zero, otherwise a precise Data abort exception is generated.

See *Unaligned access support* on page A2-42.

Notes

Data Abort   For details of the effects of the instruction if a Data Abort occurs, see Effects of data-aborted instructions on page A2-19.

Operand restrictions

If <addressing_mode> specifies base register write-back, and the same register is specified for <Rd> and <Rn>, the results are UNPREDICTABLE.

Use of R15   If register 15 is specified for <Rd>, address[1:0] must be 0b00. If not, the result is UNPREDICTABLE.

ARM/Thumb state transfers (ARM architecture version 5 and above)

If bits[1:0] of the loaded value are 0b10, the result is UNPREDICTABLE, because branches to non word-aligned addresses are not possible in ARM state.

ARM DDI 0100F-02

A4.1.24 LDRB

31	28	27	26	25	24	23	22	21	20	19	16	15	12	11	0
cond		0	1	I	P	U	1	W	1	Rn		Rd		addr_mode	

LDRB (Load Register Byte) loads a byte from memory and zero-extends the byte to a 32-bit word.

Syntax

```
LDR{<cond>}B  <Rd>, <addressing_mode>
```

where:

<cond>   Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-7. If <cond> is omitted, the AL (always) condition is used.

<Rd>   Specifies the destination register for the loaded value. If register 15 is specified for <Rd>, the result is UNPREDICTABLE.

<addressing_mode>

Is described in *Addressing Mode 2 - Load and Store Word or Unsigned Byte* on page A5-19. It determines the I, P, U, W, Rn and addr_mode bits of the instruction.

The syntax of all forms of <addressing_mode> includes a *base register* <Rn>. Some forms also specify that the instruction modifies the base register value (this is known as *base register write-back*).

Architecture version

All.

Exceptions

Data Abort.

Operation

```
if ConditionPassed(cond) then
    Rd = Memory[address,1]
```

Usage

Combined with a suitable addressing mode, LDRB allows 8-bit memory data to be loaded into a general-purpose register where it can be manipulated.

Using the PC as the base register allows PC-relative addressing, to facilitate position-independent code.
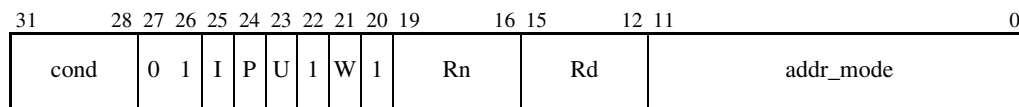

Notes

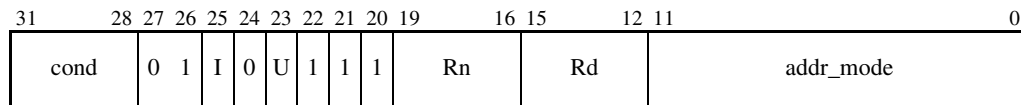**Operand restrictions**

If `<addressing_mode>` specifies base register write-back, and the same register is specified for `<Rd>` and `<Rn>`, the results are UNPREDICTABLE.

Data Abort For details of the effects of the instruction if a Data Abort occurs, see *Effects of data-aborted instructions* on page A2-21.

A4.1.25 LDRBT

31	28	27	26	25	24	23	22	21	20	19	16	15	12	11	0
cond		0	1	I	0	U	1	1	1	Rn		Rd		addr_mode	

LDRBT (Load Register Byte with Translation) loads a byte from memory and zero-extends the byte to a 32-bit word.

If LDRBT is executed when the processor is in a privileged mode, the memory system is signaled to treat the access as if the processor were in User mode.

Syntax

```
LDR{<cond>}BT  <Rd>, <post_indexed_addressing_mode>
```

where:

<cond>   Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-7. If <cond> is omitted, the AL (always) condition is used.

<Rd>   Specifies the destination register for the loaded value. If R15 is specified for <Rd>, the result is UNPREDICTABLE.

<post_indexed_addressing_mode>

Is described in *Addressing Mode 2 - Load and Store Word or Unsigned Byte* on page A5-19. It determines the I, U, Rn and addr_mode bits of the instruction. Only post-indexed forms of Addressing Mode 2 are available for this instruction. These forms have P == 0 and W == 0, where P and W are bit[24] and bit[21] respectively. This instruction uses P == 0 and W == 1 instead, but the addressing mode is the same in all other respects.

The syntax of all forms of <post_indexed_addressing_mode> includes a *base register* <Rn>. All forms also specify that the instruction modifies the base register value (this is known as *base register write-back*).

Architecture version

All.

Exceptions

Data Abort.

Operation

```
if ConditionPassed(cond) then
    Rd = Memory[address,1]
```

Usage

`LDRBT` can be used by a (privileged) exception handler that is emulating a memory access instruction that would normally execute in User mode. The access is restricted as if it had User mode privilege.
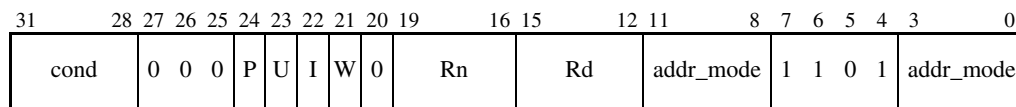
Notes

User mode   If this instruction is executed in User mode, an ordinary User mode access is performed.

Operand restrictions

If the same register is specified for `<Rd>` and `<Rn>`, the results are UNPREDICTABLE.

Data Abort   For details of the effects of the instruction if a Data Abort occurs, see *Effects of data-aborted instructions* on page A2-21.

A4.1.26   LDRD

31	28	27	26	25	24	23	22	21	20	19	16	15	12	11	8	7	6	5	4	3	0
cond		0	0	0	P	U	I	W	0	Rn		Rd		addr_mode		1	1	0	1	addr_mode	

LDRD (Load Registers Doubleword) loads a pair of ARM registers from two consecutive words of memory. The pair of registers is restricted to being an even-numbered register and the odd-numbered register that immediately follows it (for example, R10 and R11).

A greater variety of addressing modes is available than for a 2-register LDM. However, the address of the first of the two words is required to be doubleword-aligned (that is, the address must be divisible by 8).

Syntax

```
LDR{<cond>}D  <Rd>, <addressing_mode>
```

where:

<cond>   Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-7. If <cond> is omitted, the AL (always) condition is used.

<Rd>   Specifies the even-numbered destination register for the memory word addressed by <addressing_mode>. The immediately following odd-numbered register is the destination register for the next memory word. If <Rd> is R14, which would specify R15 as the second destination register, the instruction is UNPREDICTABLE. If <Rd> specifies an odd-numbered register, the instruction is UNDEFINED.

<addressing_mode>

Is described in *Addressing Mode 3 - Miscellaneous Loads and Stores* on page A5-38. It determines the P, U, I, W, Rn, and addr_mode bits of the instruction. The syntax of all forms of <addressing_mode> includes a *base register* <Rn>. Some forms also specify that the instruction modifies the base register value (this is known as *base register write-back*).

The address generated by <addressing_mode> is the address of the lower of the two words loaded by the LDRD instruction. The address of the higher word is generated by adding 4 to this address.

Architecture version

Version 6 and above, and version 5E, but not ARMv5TExP.

Exceptions

Data Abort.

Operation

```
if ConditionPassed(cond) then
    if (Rd is even-numbered) then
        if (address[2:0] == 0b000) and (Rd is not R14) then
            Rd = Memory[address,4]
            R(d+1) = Memory[address+4,4]
        else
            UNPREDICTABLE
    else
        UNDEFINED
```

Notes

Operand restrictions

If <addressing_mode> performs base register write-back and the base register <Rn> is one of the two destination registers of the instruction, the results are UNPREDICTABLE.

If <addressing_mode> specifies an index register <Rm> and <Rm> is one of the two destination registers of the instruction, the results are UNPREDICTABLE.

Data Abort For details of the effects of the instruction if a Data Abort occurs, see *Effects of data-aborted instructions* on page A2-21.

Non doubleword-aligned addresses

If the load address is not doubleword-aligned, the results are UNPREDICTABLE.

Alignment If an implementation includes a System Control coprocessor (see Chapter B3 *The System Control Coprocessor*), and alignment checking is enabled, then:

- an address with bits[1:0] != 0b00 causes an alignment exception

- an address with bits[2:0] == 0b000 does not cause an alignment exception

- it is IMPLEMENTATION DEFINED whether an address with bits[2:0] == 0b100 causes an alignment exception.

Time order The time order of the accesses to the two memory words is not architecturally defined. In particular, an implementation is allowed to perform the two 32-bit memory accesses in either order, or to combine them into a single 64-bit memory access.

A4.1.27 LDREX

31 28	27 26 25 24 23 22 21	20	19 16	15 12	11 8	7 6 5 4	3 0
cond	0 0 0 1 1 0 0	1	Rn	Rd	SBO	1 0 0 1	SBO
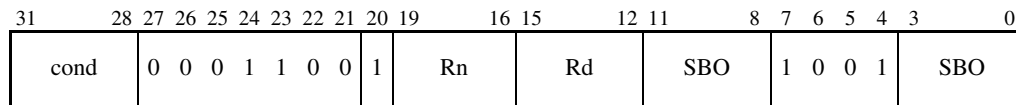
LDREX (Load Register Exclusive) loads a register from memory, and either:

- if the physical address has the Shared TLB attribute, marks the address as exclusive access for the executing processor

- otherwise, causes the executing processor to tag the fact that it has an outstanding tagged physical.

Syntax

```
LDREX{<cond>} <Rd>, [<Rn>]
```

where:

<cond>   Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-7. If `<cond>` is omitted, the AL (always) condition is used.

<Rd>   Specifies the destination register for the memory word addressed by `<Rd>`.

<Rn>   Specifies the register containing the address.

Architecture version

Version 6 and above.

Exceptions

Data Abort.

Operation

```
if ConditionPassed(cond) then
    processor_id = ExecutingProcessor()
    Rd = Memory[Rn,4]
    if Shared(Rn) == 1 then
        physical_address = TLB(Rn)
        MarkExclusiveGlobal(physical_address,processor_id)
    else
        MarkExclusiveLocal(processor_id)
```

Usage

Use LDREX in combination with STREX to implement inter-process communication in shared memory multiprocessor systems. For more information see *Synchronization primitives* on page A2-48.

Notes

Use of R15   Specifying R15 for register <Rd> or <Rn> has UNPREDICTABLE results.

**Data Abort**   For details of the effects of the instruction if a Data Abort occurs, see *Effects of data-aborted instructions* on page A2-21. If a Data Abort occurs during execution of a LDREX instruction, the MarkExclusiveGlobal() and MarkExclusiveLocal() operations are not executed and <Rd> is not updated.

Alignment   If <Rn>[1:0] != 0b00, then:

- If the implementation includes a System Control coprocessor (see Chapter B3 *The System Control Coprocessor*), and alignment checking is enabled, an alignment exception occurs.

- Otherwise, the result is UNPREDICTABLE.

A4.1.28 LDRH

31	28	27	26	25	24	23	22	21	20	19	16	15	12	11	8	7	6	5	4	3	0
cond		0	0	0	P	U	I	W	1	Rn		Rd		addr_mode		1	0	1	1	addr_mode	
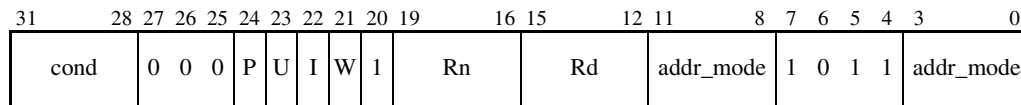
LDRH (Load Register Halfword) loads a halfword from memory and zero-extends it to a 32-bit word. If the address is not halfword-aligned, the result is UNPREDICTABLE.

Syntax

```
LDR{<cond>}H  <Rd>, <addressing_mode>
```

where:

`<cond>`	Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-7. If `<cond>` is omitted, the AL (always) condition is used.
`<Rd>`	Specifies the destination register for the loaded value. If R15 is specified for `<Rd>`, the result is UNPREDICTABLE.

`<addressing_mode>`

Is described in *Addressing Mode 3 - Miscellaneous Loads and Stores* on page A5-38. It determines the P, U, I, W, Rn and addr_mode bits of the instruction.

The syntax of all forms of `<addressing_mode>` includes a *base register* `<Rn>`. Some forms also specify that the instruction modifies the base register value (this is known as *base register write-back*).

Architecture version

Version 4 and above.

Exceptions

Data Abort.

Operation

```
if ConditionPassed(cond) then
    if address[0] == 0
        data = Memory[address,2]
    else /* address[0] == 1 */
        data = UNPREDICTABLE
    Rd = data
```

Usage

Used with a suitable addressing mode, LDRH allows 16-bit memory data to be loaded into a general-purpose register where its value can be manipulated.

Using the PC as the base register allows PC-relative addressing to facilitate position-independent code.

Notes

Operand restrictions

If <addressing_mode> specifies base register write-back, and the same register is specified for <Rd> and <Rn>, the results are UNPREDICTABLE.

Data Abort   For details of the effects of the instruction if a Data Abort occurs, see *Effects of data-aborted instructions* on page A2-21.

Non halfword-aligned addresses

If the load address is not halfword-aligned, the loaded value is UNPREDICTABLE.

Alignment   If an implementation includes a System Control coprocessor (see Chapter B3 *The System Control Coprocessor*), and alignment checking is enabled, an address with bit[0] != 0 causes an alignment exception.

A4.1.29 LDRSB

31	28	27	26	25	24	23	22	21	20	19	16	15	12	11	8	7	6	5	4	3	0
cond		0	0	0	P	U	I	W	1	Rn		Rd		addr_mode		1	1	0	1	addr_mode	

LDRSB (Load Register Signed Byte) loads a byte from memory and sign-extends the byte to a 32-bit word.

Syntax

```
LDR{<cond>}SB  <Rd>, <addressing_mode>
```

where:

<cond>   Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-7. If <cond> is omitted, the AL (always) condition is used.

<Rd>   Specifies the destination register for the loaded value. If R15 is specified for <Rd>, the result is UNPREDICTABLE.

<addressing_mode>

                    Is described in *Addressing Mode 3 - Miscellaneous Loads and Stores* on page A5-38. It determines the P, U, I, W, Rn and addr_mode bits of the instruction.

                    The syntax of all forms of <addressing_mode> includes a *base register* <Rn>. Some forms also specify that the instruction modifies the base register value (this is known as *base register write-back*).

Architecture version

Version 4 and above.

Exceptions

Data Abort.

Operation

```
if ConditionPassed(cond) then
    data = Memory[address,1]
    Rd = SignExtend(data)
```

Usage

Use LDRSB with a suitable addressing mode to load 8-bit signed memory data into a general-purpose register where it can be manipulated.

You can perform PC-relative addressing by using the PC as the base register. This facilitates position-independent code.

Notes

Operand restrictions

> If <addressing_mode> specifies base register write-back, and the same register is specified for <Rd> and <Rn>, the results are UNPREDICTABLE.
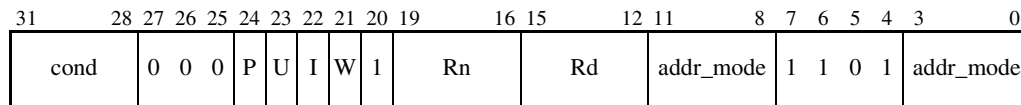
Data Abort   For details of the effects of the instruction if a Data Abort occurs, see *Effects of data-aborted instructions* on page A2-21.

A4.1.30 LDRSH

31	28	27	26	25	24	23	22	21	20	19	16	15	12	11	8	7	6	5	4	3	0
cond		0	0	0	P	U	I	W	1	Rn		Rd		addr_mode		1	1	1	1	addr_mode	

LDRSH (Load Register Signed Halfword) loads a halfword from memory and sign-extends the halfword to a 32-bit word.

If the address is not halfword-aligned, the result is UNPREDICTABLE.

Syntax

```
LDR{<cond>}SH  <Rd>, <addressing_mode>
```

where:

<cond>   Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-7. If <cond> is omitted, the AL (always) condition is used.

<Rd>   Specifies the destination register for the loaded value. If R15 is specified for <Rd>, the result is UNPREDICTABLE.

<addressing_mode>

Is described in *Addressing Mode 3 - Miscellaneous Loads and Stores* on page A5-38. It determines the P, U, I, W, Rn and addr_mode bits of the instruction.

The syntax of all forms of <addressing_mode> includes a *base register* <Rn>. Some forms also specify that the instruction modifies the base register value (this is known as *base register write-back*).

Architecture version

Version 4 and above.

Exceptions

Data Abort.

Operation

```
if ConditionPassed(cond) then
    if address[0] == 0
        data = Memory[address,2]
    else /* address[0] == 1 */
        data = UNPREDICTABLE
    Rd = SignExtend(data)
```

Usage

Used with a suitable addressing mode, LDRSH allows 16-bit signed memory data to be loaded into a general-purpose register where its value can be manipulated.

Using the PC as the base register allows PC-relative addressing, which facilitates position-independent code.
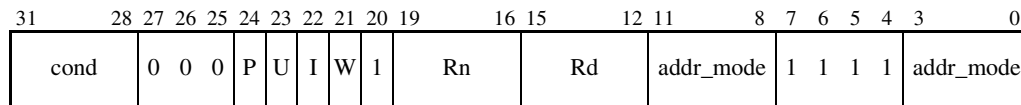
Notes

Operand restrictions

If <addressing_mode> specifies base register write-back, and the same register is specified for <Rd> and <Rn>, the results are UNPREDICTABLE.

Data Abort   For details of the effects of the instruction if a Data Abort occurs, see *Effects of data-aborted instructions* on page A2-21.

Non halfword-aligned addresses

If the load address is not halfword-aligned, the loaded value is UNPREDICTABLE.

Alignment   If an implementation includes a System Control coprocessor (see Chapter B3 *The System Control Coprocessor*), and alignment checking is enabled, an address with bit[0] != 0 causes an alignment exception.

A4.1.31 LDRT

31	28	27	26	25	24	23	22	21	20	19	16	15	12	11	0
cond		0	1	I	0	U	0	1	1	Rn		Rd		addr_mode	
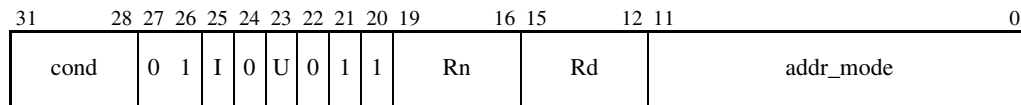
LDRT (Load Register with Translation) loads a word from memory.

If LDRT is executed when the processor is in a privileged mode, the memory system is signaled to treat the access as if the processor were in User mode.

Syntax

LDR{<cond>}T <Rd>, <post_indexed_addressing_mode>

where:

<cond>   Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-7. If <cond> is omitted, the AL (always) condition is used.

<Rd>   Specifies the destination register for the loaded value. If R15 is specified for <Rd>, the result is UNPREDICTABLE.

<post_indexed_addressing_mode>

Is described in *Addressing Mode 2 - Load and Store Word or Unsigned Byte* on page A5-19. It determines the I, U, Rn and addr_mode bits of the instruction. Only post-indexed forms of Addressing Mode 2 are available for this instruction. These forms have P == 0 and W == 0, where P and W are bit[24] and bit[21] respectively. This instruction uses P == 0 and W == 1 instead, but the addressing mode is the same in all other respects.

The syntax of all forms of <post_indexed_addressing_mode> includes a *base register* <Rn>. All forms also specify that the instruction modifies the base register value (this is known as *base register write-back*).

Architecture version

All.

Exceptions

Data Abort.

Operation

```
if ConditionPassed(cond) then
    if address[1:0] == 0b00
        Rd = Memory[address,4]
    else if address[1:0] == 0b01
        Rd = Memory[address,4] Rotate_Right 8
    else if address[1:0] == 0b10
        Rd = Memory[address,4] Rotate_Right 16
    else /* address[1:0] == 0b11 */
        Rd = Memory[address,4] Rotate_Right 24
```

Usage

LDRT can be used by a (privileged) exception handler that is emulating a memory access instruction that would normally execute in User mode. The access is restricted as if it had User mode privilege.

Notes

User mode If this instruction is executed in User mode, an ordinary User mode access is performed.

Operand restrictions

If the same register is specified for <Rd> and <Rn> the results are UNPREDICTABLE.

Data Abort For details of the effects of the instruction if a Data Abort occurs, see *Effects of data-aborted instructions* on page A2-21.

Alignment As for LDR, see *LDR* on page A4-42.

A4.1.32 MCR

31	28	27	26	25	24	23	21	20	19	16	15	12	11	8	7	5	4	3	0
cond		1	1	1	0	opcode_1		0	CRn		Rd		cp_num		opcode_2		1	CRm	

MCR (Move to Coprocessor from ARM Register) passes the value of register `<Rd>` to the coprocessor whose number is cp_num.

If no coprocessors indicate that they can execute the instruction, an Undefined Instruction exception is generated.

Syntax

```
MCR{<cond>}  <coproc>, <opcode_1>, <Rd>, <CRn>, <CRm>{, <opcode_2>}
MCR2         <coproc>, <opcode_1>, <Rd>, <CRn>, <CRm>{, <opcode_2>}
```

where:

`<cond>`   Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-7. If `<cond>` is omitted, the AL (always) condition is used.

`MCR2`   Causes the condition field of the instruction to be set to 0b1111. This provides additional opcode space for coprocessor designers. The resulting instructions can only be executed unconditionally.

`<coproc>`   Specifies the name of the coprocessor, and causes the corresponding coprocessor number to be placed in the cp_num field of the instruction. The standard generic coprocessor names are p0, p1, ..., p15.

`<opcode_1>` Is a coprocessor-specific opcode.

`<Rd>`   Is the ARM register whose value is transferred to the coprocessor. If R15 is specified for `<Rd>`, the result is UNPREDICTABLE.

`<CRn>`   Is the destination coprocessor register.

`<CRm>`   Is an additional destination or source coprocessor register.

`<opcode_2>` Is a coprocessor-specific opcode. If it is omitted, `<opcode_2>` is assumed to be 0.

Architecture version

MCR is in version 2 and above.

MCR2 is in version 5 and above.

Exceptions

Undefined Instruction.

Operation

```
if ConditionPassed(cond) then
    send Rd value to Coprocessor[cp_num]
```

Usage

Use MCR to initiate a coprocessor operation that acts on a value from an ARM register. An example is a fixed-point to floating-point conversion instruction for a floating-point coprocessor.

Notes

Coprocessor fields   Only instruction bits[31:24], bit[20], bits[15:8], and bit[4] are defined by the ARM architecture. The remaining fields are recommendations, for compatibility with ARM Development Systems.

Unimplemented coprocessor instructions

Hardware coprocessor support is optional, regardless of the architecture version. An implementation can choose to implement a subset of the coprocessor instructions, or no coprocessor instructions at all. Any coprocessor instructions that are not implemented instead cause an undefined instruction exception.

A4.1.33 MCRR

31	28	27	26	25	24	23	22	21	20	19	16	15	12	11	8	7	4	3	0
cond		1	1	0	0	0	1	0	0	Rn		Rd		cp_num		opcode		CRm	

MCRR (Move to Coprocessor from two ARM Registers) passes the values of two ARM registers to a coprocessor.

If no coprocessors indicate that they can execute the instruction, an undefined instruction exception is generated.

Syntax

```
MCRR{<cond>}  <coproc>, <opcode>, <Rd>, <Rn>, <CRm>
MCRR2  <coproc>, <opcode>, <Rd>, <Rn>, <CRm>
```

where:

<cond>	Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-7. If <cond> is omitted, the AL (always) condition is used.
MCRR2	Causes the condition field of the instruction to be set to 0b1111. This provides additional opcode space for coprocessor designers. The resulting instructions can only be executed unconditionally.
<coproc>	Specifies the name of the coprocessor, and causes the corresponding coprocessor number to be placed in the cp_num field of the instruction. The standard generic coprocessor names are p0, p1, …, p15.
<opcode>	Is a coprocessor-specific opcode.
<Rd>	Is the first ARM register whose value is transferred to the coprocessor. If R15 is specified for <Rd>, the result is UNPREDICTABLE.
<Rn>	Is the second ARM register whose value is transferred to the coprocessor. If R15 is specified for <Rn>, the result is UNPREDICTABLE.
<CRm>	Is the destination coprocessor register.

Architecture version

MCRR is in version 6 and above, and version 5E, but not ARMv5TExP.

MCRR2 is in version 6 and above.

Exceptions

Undefined instruction.

Operation

```
if ConditionPassed(cond) then
    send Rd value to Coprocessor[cp_num]
    send Rn value to Coprocessor[cp_num]
```

Usage

Use MCRR to initiate a coprocessor operation that acts on values from two ARM registers. An example for a floating-point coprocessor is an instruction to transfer a double-precision floating-point number held in two ARM registers to a floating-point register.

Notes

Coprocessor fields

Only instruction bits[31:8] are defined by the ARM architecture. The remaining fields are recommendations, for compatibility with ARM Development Systems.

Unimplemented coprocessor instructions

Hardware coprocessor support is optional, regardless of the architecture version. An implementation can choose to implement a subset of the coprocessor instructions, or no coprocessor instructions at all. Any coprocessor instructions that are not implemented instead cause an undefined instruction exception.

Order of transfers

If a coprocessor uses these instructions, it defines how each of the values of <Rd> and <Rn> is used. There is no architectural requirement for the two register transfers to occur in any particular time order. It is IMPLEMENTATION DEFINED whether Rd is transferred before Rn, after Rn, or at the same time as Rn.

A4.1.34 MLA

31	28	27 26 25 24 23 22 21	20	19	16	15	12	11	8	7 6 5 4	3	0
cond		0 0 0 0 0 0 1	S	Rd		Rn		Rs		1 0 0 1	Rm	
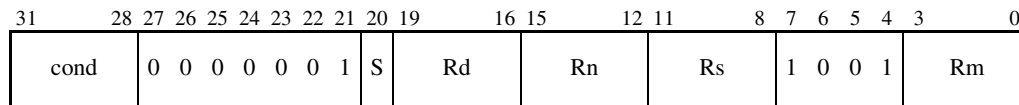
MLA (Multiply Accumulate) multiplies two signed or unsigned 32-bit values, and adds a third 32-bit value. The least significant 32 bits of the result are written to the destination register.

MLA can optionally update the condition code flags, based on the result.

Syntax

MLA{<cond>}{S} <Rd>, <Rm>, <Rs>, <Rn>

where:

<cond>   Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-7. If <cond> is omitted, the AL (always) condition is used.

S   Causes the S bit (bit[20]) in the instruction to be set to 1 and specifies that the instruction updates the CPSR by setting the N and Z flags according to the result of the multiply-accumulate. If S is omitted, the S bit of the instruction is set to 0 and the entire CPSR is unaffected by the instruction.

<Rd>   Specifies the destination register.

<Rm>   Holds the value to be multiplied with the value of <Rs>.

<Rs>   Holds the value to be multiplied with the value of <Rm>.

<Rn>   Contains the value that is added to the product of <Rs> and <Rm>.

Architecture version

Version 2 and above.

Exceptions

None.

Operation

```
if ConditionPassed(cond) then
    Rd = (Rm * Rs + Rn)[31:0]
    if S == 1 then
        N Flag = Rd[31]
        Z Flag = if Rd == 0 then 1 else 0
        C Flag = unaffected in v5 and above, UNPREDICTABLE in v4 and earlier
        V Flag = unaffected
```

Notes

Use of R15 Specifying R15 for register <Rd>, <Rm>, <Rs>, or <Rn> has UNPREDICTABLE results.

Early termination If the multiplier implementation supports early termination, it must be implemented on the value of the <Rs> operand. The type of early termination used (signed or unsigned) is IMPLEMENTATION DEFINED.

Signed and unsigned Because the MLA instruction produces only the lower 32 bits of the 64-bit product, MLA gives the same answer for multiplication of both signed and unsigned numbers.

C flag The MLAS instruction is defined to leave the C flag unchanged in ARM architecture version 5 and above. In earlier versions of the architecture, the value of the C flag was UNPREDICTABLE after a MLAS instruction.

Operand restriction Specifying the same register for <Rd> and <Rm> was previously described as producing UNPREDICTABLE results. There is no restriction in ARMv6, and it is believed all relevant ARMv4 and ARMv5 implementations do not require this restriction either, because high performance multipliers read all their operands prior to writing back any results.

ARM DDI 0100F-02

A4.1.35 MOV

31	28	27 26	25	24 23 22 21	20	19 16	15 12	11 0
cond		0 0	I	1 1 0 1	S	SBZ	Rd	shifter_operand

MOV (Move) writes a value to the destination register. The value can be either an immediate value or a value from a register, and can be shifted before the write.

MOV can optionally update the condition code flags, based on the result.

Syntax

MOV{<cond>}{S} <Rd>, <shifter_operand>

where:

<cond>   Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-7. If <cond> is omitted, the AL (always) condition is used.

S   Sets the S bit (bit[20]) in the instruction to 1 and specifies that the instruction updates the CPSR. If S is omitted, the S bit is set to 0 and the CPSR is not changed by the instruction. Two types of CPSR update can occur when S is specified:

- If <Rd> is not R15, the N and Z flags are set according to the value moved (post-shift if a shift is specified), and the C flag is set to the carry output bit generated by the shifter (see *Addressing Mode 1 - Data-processing operands* on page A5-2). The V flag and the rest of the CPSR are unaffected.

- If <Rd> is R15, the SPSR of the current mode is copied to the CPSR. This form of the instruction is UNPREDICTABLE if executed in User mode or System mode, because these modes do not have an SPSR.

<Rd>   Specifies the destination register.

<shifter_operand>

Specifies the operand. The options for this operand are described in *Addressing Mode 1 - Data-processing operands* on page A5-2, including how each option causes the I bit (bit[25]) and the shifter_operand bits (bits[11:0]) to be set in the instruction.

If the I bit is 0 and both bit[7] and bit[4] of shifter_operand are 1, the instruction is not MOV. Instead, see *Extending the instruction set* on page A3-35 to determine which instruction it is.

Architecture version

All.

Exceptions

None.

Operation

```
if ConditionPassed(cond) then
    Rd = shifter_operand
    if S == 1 and Rd == R15 then
        if CurrentModeHasSPSR() then
            CPSR = SPSR
        else UNPREDICTABLE
    else if S == 1 then
        N Flag = Rd[31]
        Z Flag = if Rd == 0 then 1 else 0
        C Flag = shifter_carry_out
        V Flag = unaffected
```
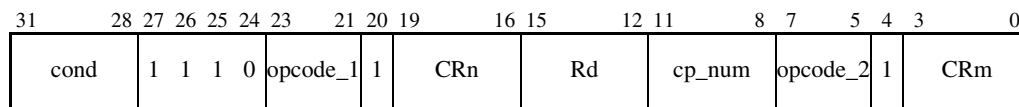
Usage

Use MOV to:

- Move a value from one register to another.
- Put a constant value into a register.
- Perform a shift without any other arithmetic or logical operation. Use a left shift by *n* to multiply by 2^n.
- When the PC is the destination of the instruction, a branch occurs. The instruction:

      MOV PC, LR

  can therefore be used to return from a subroutine (see instructions *B, BL* on page A4-10). In T variants of architecture 4 and in architecture 5 and above, the instruction BX LR must be used in place of MOV PC, LR, as the BX instruction automatically switches back to Thumb state if appropriate (but see also *The T bit* on page A2-13 for operation on non-T variants of ARM architecture version 5).
- When the PC is the destination of the instruction and the S bit is set, a branch occurs and the SPSR of the current mode is copied to the CPSR. This means that you can use a MOVS PC, LR instruction to return from some types of exception (see *Exceptions* on page A2-15).

A4.1.36 MRC

31	28	27	26	25	24	23	21	20	19	16	15	12	11	8	7	5	4	3	0
cond		1	1	1	0	opcode_1		1	CRn		Rd		cp_num		opcode_2		1	CRm	

MRC (Move to ARM Register from Coprocessor) causes a coprocessor to transfer a value to an ARM register or to the condition flags.

If no coprocessors can execute the instruction, an Undefined Instruction exception is generated.

Syntax

```
MRC{<cond>}   <coproc>, <opcode_1>, <Rd>, <CRn>, <CRm>{, <opcode_2>}
MRC2          <coproc>, <opcode_1>, <Rd>, <CRn>, <CRm>{, <opcode_2>}
```

where:

<cond>   Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-7. If <cond> is omitted, the AL (always) condition is used.

MRC2   Causes the condition field of the instruction to be set to 0b1111. This provides additional opcode space for coprocessor designers. The resulting instructions can only be executed unconditionally.

<coproc>   Specifies the name of the coprocessor, and causes the corresponding coprocessor number to be placed in the cp_num field of the instruction. The standard generic coprocessor names are p0, p1, ..., p15.

<opcode_1>   Is a coprocessor-specific opcode.

<Rd>   Specifies the destination ARM register for the instruction. If R15 is specified for <Rd>, the condition code flags are updated instead of a general-purpose register.

<CRn>   Specifies the coprocessor register that contains the first operand.

<CRm>   Is an additional coprocessor source or destination register.

<opcode_2>   Is a coprocessor-specific opcode. If it is omitted, <opcode_2> is assumed to be 0.

Architecture version

MRC is in version 2 and above.

MRC2 is in version 5 and above.

Exceptions

Undefined Instruction.

Operation

```
if ConditionPassed(cond) then
    data = value from Coprocessor[cp_num]
    if Rd is R15 then
        N flag = data[31]
        Z flag = data[30]
        C flag = data[29]
        V flag = data[28]
    else /* Rd is not R15 */
        Rd = data
```

Usage

MRC has two uses:

1.   If <Rd> specifies R15, the condition code flags bits are updated from the top four bits of the value from the coprocessor specified by <coproc> (to allow conditional branching on the status of a coprocessor) and the other 28 bits are ignored.

     An example of this use would be to transfer the result of a comparison performed by a floating-point coprocessor to the ARM's condition flags.

2.   Otherwise the instruction writes into register <Rd> a value from the coprocessor specified by <coproc>. TBD can have an operation as well.

     An example of this use is a floating-point to integer conversion instruction in a floating-point coprocessor.

Notes

Coprocessor fields   Only instruction bits[31:24], bit[20], bits[15:8] and bit[4] are defined by the ARM architecture. The remaining fields are recommendations, for compatibility with ARM Development Systems.

Unimplemented coprocessor instructions

                              Hardware coprocessor support is optional, regardless of the architecture version. An implementation can choose to implement a subset of the coprocessor instructions, or no coprocessor instructions at all. Any coprocessor instructions that are not implemented instead cause an undefined instruction exception.

A4.1.37 MRRC

31 28	27 26 25 24 23 22 21 20	19 16	15 12	11 8	7 4	3 0
cond	1 1 0 0 0 1 0 1	Rn	Rd	cp_num	opcode	CRm

MRRC (Move to two ARM Registers from Coprocessor) causes a coprocessor to transfer values to two ARM registers.

If no coprocessors can execute the instruction, an undefined instruction exception is generated. TBD look for similar and make match.

Syntax

```
MRRC{<cond>}  <coproc>, <opcode>, <Rd>, <Rn>, <CRm>
MRRC2  <coproc>, <opcode>, <Rd>, <Rn>, <CRm>
```

where:

<cond>   Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-7. If <cond> is omitted, the AL (always) condition is used.

MRRC2   Causes the condition field of the instruction to be set to 0b1111. This provides additional opcode space for coprocessor designers. The resulting instructions can only be executed unconditionally.

<coproc>   Specifies the name of the coprocessor, and causes the corresponding coprocessor number to be placed in the cp_num field of the instruction. The standard generic coprocessor names are p0, p1, …, p15.

<opcode>   Is a coprocessor-specific opcode.

<Rd>   Is the first destination ARM register. If R15 is specified for <Rd>, the result is UNPREDICTABLE.

<Rn>   Is the second destination ARM register. If R15 is specified for <Rn>, the result is UNPREDICTABLE.

<CRm>   Is the coprocessor register which supplies the data to be transferred.

Architecture version

MRRC is in version 6 and above, and version 5E, but not ARMv5TExP.

MRRC2 is in version 6 and above.

Exceptions

Undefined instruction.

Operation

```
if ConditionPassed(cond) then
    Rd = first value from Coprocessor[cp_num]
    Rn = second value from Coprocessor[cp_num]
```

Usage

Use `MRRC` to initiate a coprocessor operation that writes values to two ARM registers. An example for a floating-point coprocessor is an instruction to transfer a double-precision floating-point number held in a floating-point register to two ARM registers.

Notes

Operand restrictions

Specifying the same register for `<Rd>` and `<Rn>` has UNPREDICTABLE results.

Coprocessor fields

Only instruction bits[31:8] are defined by the ARM architecture. The remaining fields are recommendations, for compatibility with ARM Development Systems.

Unimplemented coprocessor instructions

Hardware coprocessor support is optional, regardless of the architecture version. An implementation can choose to implement a subset of the coprocessor instructions, or no coprocessor instructions at all. Any coprocessor instructions that are not implemented instead cause an undefined instruction exception.

Order of transfers

If a coprocessor uses these instructions, it defines which value is written to `<Rd>` and which value to `<Rn>`. There is no architectural requirement for the two register transfers to occur in any particular time order. It is IMPLEMENTATION DEFINED whether Rd is transferred before Rn, after Rn, or at the same time as Rn.

A4.1.38 MRS

31	28	27	26	25	24	23	22	21	20	19	16	15	12	11	0
cond		0	0	0	1	0	R	0	0	SBO		Rd		SBZ	
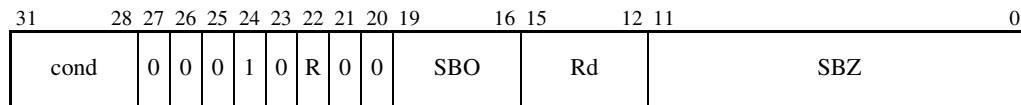
MRS (Move PSR to general-purpose Register) moves the value of the CPSR or the SPSR of the current mode into a general-purpose register. In the general-purpose register, the value can be examined or manipulated with normal data-processing instructions.

Syntax

```
MRS{<cond>}  <Rd>, CPSR
MRS{<cond>}  <Rd>, SPSR
```

where:

<cond>   Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-7. If <cond> is omitted, the AL (always) condition is used.

<Rd>   Specifies the destination register. If R15 is specified for <Rd>, the result is UNPREDICTABLE.

Architecture version

Version 3 and above.

Exceptions

None.

Operation

```
if ConditionPassed(cond) then
    if R == 1 then
        Rd = SPSR
    else
        Rd = CPSR
```

Usage

The MRS instruction is commonly used for three purposes:

- As part of a read/modify/write sequence for updating a PSR. For more details, see *MSR* on page A4-75.
- When an exception occurs and there is a possibility of a nested exception of the same type occurring, the SPSR of the exception mode is in danger of being corrupted. To deal with this, the SPSR value must be saved before the nested exception can occur, and later restored in preparation for the exception return. The saving is normally done by using an MRS instruction followed by a store instruction. Restoring the SPSR uses the reverse sequence of a load instruction followed by an MSR instruction.
- In process swap code, the programmer's model state of the process being swapped out must be saved, including relevant PSR contents, and similar state of the process being swapped in must be restored. Again, this involves the use of MRS/store and load/MSR instruction sequences.
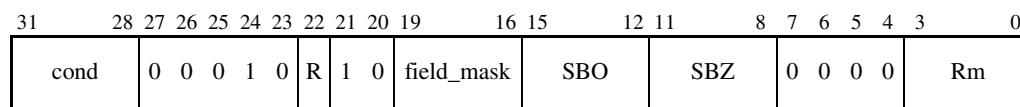
Notes

Opcode [11:0]   Execution of MRS instructions with opcode[11:0] != 0x000 is UNPREDICTABLE.

Opcode [19:16]   Execution of MRS instructions with opcode[19:16] != 0b1111 is UNPREDICTABLE.

User mode SPSR   Accessing the SPSR when in User mode or System mode is UNPREDICTABLE.

A4.1.39 MSR

Immediate operand:

31 28	27 26 25 24 23	22	21 20	19 16	15 12	11 8	7 0
cond	0 0 1 1 0	R	1 0	field_mask	SBO	rotate_imm	8_bit_immediate

Register operand:

31 28	27 26 25 24 23	22	21 20	19 16	15 12	11 8	7 6 5 4	3 0
cond	0 0 0 1 0	R	1 0	field_mask	SBO	SBZ	0 0 0 0	Rm
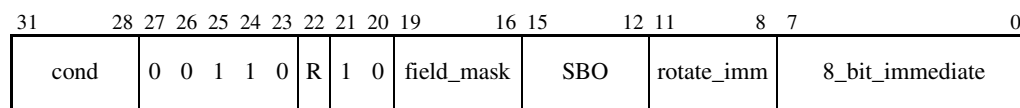
`MSR` (Move to Status Register from ARM Register) transfers the value of a general-purpose register or an immediate constant to the CPSR or the SPSR of the current mode.

Syntax

```
MSR{<cond>}  CPSR_<fields>, #<immediate>
MSR{<cond>}  CPSR_<fields>, <Rm>
MSR{<cond>}  SPSR_<fields>, #<immediate>
MSR{<cond>}  SPSR_<fields>, <Rm>
```

where:

`<cond>`	Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-7. If `<cond>` is omitted, the `AL` (always) condition is used.
`<fields>`	Is a sequence of one or more of the following:
	`c` sets the control field mask bit (bit 16)
	`x` sets the extension field mask bit (bit 17)
	`s` sets the status field mask bit (bit 18)
	`f` sets the flags field mask bit (bit 19).
`<immediate>`	Is the immediate value to be transferred to the CPSR or SPSR. Allowed immediate values are 8-bit immediates (in the range `0x00` to `0xFF`) and values that can be obtained by rotating them right by an even amount in the range 0–30. These immediate values are the same as those allowed in the immediate form as shown in *Data-processing operands - Immediate* on page A5-6.
`<Rm>`	Is the general-purpose register to be transferred to the CPSR or SPSR.

Architecture version

Version 3 and above.

Exceptions

None.

Operation

There are four categories of PSR bits, according to rules about updating them, see *Types of PSR bits* on page A2-10 for details.

The pseudo-code uses four bit mask constants to identify these categories of PSR bits. The values of these masks depend on the architecture version, see Table A4-1.

Table A4-1 Bit mask constants

Architecture versions[a]	UnallocMask	UserMask	PrivMask	StateMask
3, 3G, 3M, 4, 4xM, 5, 5xM	0x0FFFFF20	0xF0000000	0x0000000F	0x00000000
4T, 4TxM, 5T, 5TxM	0x0FFFFF00	0xF0000000	0x0000000F	0x00000020
5TE, 5TExP	0x07FFFF00	0xF8000000	0x0000000F	0x00000020
5TEJ	0x06FFFF00	0xF8000000	0x0000000F	0x01000020
6	0x06F0FC00	0xF80F0200	0x000001DF	0x01000020

    a. Architecture variants shown for completeness. All version 3 variants are now OBSOLETE.

```
if ConditionPassed(cond) then
    if opcode[25] == 1
        operand = 8_bit_immediate Rotate_Right (rotate_imm * 2)
    else
        operand = Rm
    if (operand AND UnallocMask) !=0 then
        UNPREDICTABLE              /* Attempt to set reserved bits */
    byte_mask = (if field_mask[0] == 1 then 0x000000FF else 0x00000000) OR
                (if field_mask[1] == 1 then 0x0000FF00 else 0x00000000) OR
                (if field_mask[2] == 1 then 0x00FF0000 else 0x00000000) OR
                (if field_mask[3] == 1 then 0xFF000000 else 0x00000000)
    if R == 0 then
        if InAPrivilegedMode() then
            if (operand AND StateMask) != 0 then
                UNPREDICTABLE      /* Attempt to set non-ARM execution state */
            else
                mask = byte_mask AND (UserMask OR PrivMask)
        else        /* R == 1
            mask = byte_mask AND UserMask
        CPSR = (CPSR AND NOT mask) OR (operand AND mask)
    else
        if CurrentModeHasSPSR() then
            mask = byte_mask AND (UserMask OR PrivMask OR StateMask)
```

```
                        SPSR = (SPSR AND NOT mask) OR (operand AND mask)
                else
                    UNPREDICTABLE
```

Usage

Use `MSR` to update the value of the condition code flags, interrupt enables, or the processor mode.

You must normally update the value of a PSR by moving the PSR to a general-purpose register (using the `MRS` instruction), modifying the relevant bits of the general-purpose register, and restoring the updated general-purpose register value back into the PSR (using the `MSR` instruction). For example, a good way to switch the ARM to Supervisor mode from another privileged mode is:

```
    MRS   R0,CPSR                ; Read CPSR
    BIC   R0,R0,#0x1F            ; Modify by removing current mode
    ORR   R0,R0,#0x13            ; and substituting Supervisor mode
    MSR   CPSR_c,R0              ; Write the result back to CPSR
```

For maximum efficiency, `MSR` instructions should only write to those fields that they can potentially change. For example, the last instruction in the above code can only change the CPSR control field, as all bits in the other fields are unchanged since they were read from the CPSR by the first instruction. So it writes to CPSR_c, not CPSR_fsxc or some other combination of fields.

However, if the *only* reason that an `MSR` instruction cannot change a field is that no bits are currently allocated to the field, then the field must be written, to ensure future compatibility.

You can use the immediate form of `MSR` to set any of the fields of a PSR, but you must take care to use the read-modify-write technique described above. The immediate form of the instruction is equivalent to reading the PSR concerned, replacing all the bits in the fields concerned by the corresponding bits of the immediate constant and writing the result back to the PSR. The immediate form must therefore only be used when the intention is to modify all the bits in the specified fields and, in particular, must not be used if the specified fields include any as-yet-unallocated bits. Failure to observe this rule might result in code which has unanticipated side effects on future versions of the ARM architecture.

As an exception to the above rule, it is legitimate to use the immediate form of the instruction to modify the flags byte, despite the fact that bits[26:25] of the PSRs have no allocated function at present. For example, you can use `MSR` to set all four flags (and clear the Q flag if the processor implements the Enhanced DSP extension):

```
    MSR   CPSR_f,#0xF0000000
```

Any functionality allocated to bits[26:24] in a future version of the ARM architecture will be designed so that such code does not have unexpected side effects.

Notes

The R bit   Bit[22] of the instruction is 0 if the CPSR is to be written and 1 if the SPSR is to be written.

User mode CPSR

> Any writes to CPSR[23:0] in User mode are ignored (so that User mode programs cannot change to a privileged mode).

User mode SPSR

> Accessing the SPSR when in User mode is UNPREDICTABLE.

System mode SPSR

> Accessing the SPSR when in System mode is UNPREDICTABLE.

Obsolete field specification

> The CPSR, CPSR_flg, CPSR_ctl, CPSR_all, SPSR, SPSR_flg, SPSR_ctl and SPSR_all forms of PSR field specification have been superseded by the csxf format shown on page A4-75.
>
> CPSR, SPSR, CPSR_all and SPSR_all produce a field mask of 0b1001.
>
> CPSR_flg and SPSR_flg produce a field mask of 0b1000.
>
> CPSR_ctl and SPSR_ctl produce a field mask of 0b0001.

The T bit or J bit

> The MSR instruction must not be used to alter the T bit or the J bit in the CPSR. If such an attempt is made, the results are UNPREDICTABLE.

Addressing modes

> The immediate and register forms are specified in precisely the same way as the immediate and unshifted register forms of Addressing Mode 1 (see *Addressing Mode 1 - Data-processing operands* on page A5-2). All other forms of Addressing Mode 1 yield UNPREDICTABLE results.

A4.1.40 MUL

31	28	27 26 25 24 23 22 21	20	19	16	15	12	11	8	7 6 5 4	3	0
cond		0 0 0 0 0 0 0	S	Rd		SBZ		Rs		1 0 0 1	Rm	
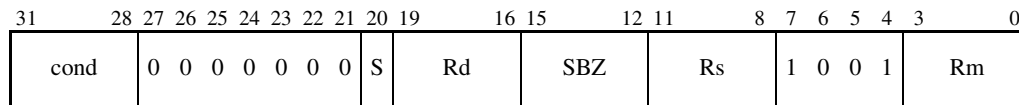
`MUL` (Multiply) multiplies two signed or unsigned 32-bit values. The least significant 32 bits of the result are written to the destination register.

`MUL` can optionally update the condition code flags, based on the result.

Syntax

```
MUL{<cond>}{S}  <Rd>, <Rm>, <Rs>
```

where:

`<cond>`	Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-7. If `<cond>` is omitted, the `AL` (always) condition is used.
`S`	Causes the S bit (bit[20]) in the instruction to be set to 1 and specifies that the instruction updates the CPSR by setting the N and Z flags according to the result of the multiplication. If `S` is omitted, the S bit of the instruction is set to 0 and the entire CPSR is unaffected by the instruction.
`<Rd>`	Specifies the destination register for the instruction.
`<Rm>`	Specifies the register that contains the first value to be multiplied.
`<Rs>`	Holds the value to be multiplied with the value of `<Rm>`.

Architecture version

Version 2 and above.

Exceptions

None.

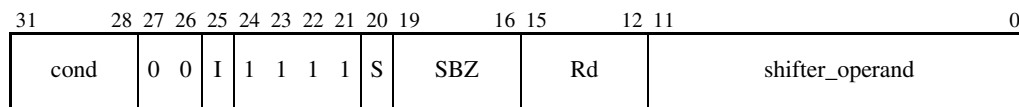### Operation

```
if ConditionPassed(cond) then
    Rd = (Rm * Rs)[31:0]
    if S == 1 then
        N Flag = Rd[31]
        Z Flag = if Rd == 0 then 1 else 0
        C Flag = unaffected in v5 and above, UNPREDICTABLE in v4 and earlier
        V Flag = unaffected
```

Notes

Use of R15   Specifying R15 for register `<Rd>`, `<Rm>`, or `<Rs>` has UNPREDICTABLE results.

Early termination   If the multiplier implementation supports early termination, it must be implemented on the value of the `<Rs>` operand. The type of early termination used (signed or unsigned) is IMPLEMENTATION DEFINED.

Signed and unsigned Because the `MUL` instruction produces only the lower 32 bits of the 64-bit product, `MUL` gives the same answer for multiplication of both signed and unsigned numbers.

C flag   The `MULS` instruction is defined to leave the C flag unchanged in ARM architecture version 5 and above. In earlier versions of the architecture, the value of the C flag was UNPREDICTABLE after a `MULS` instruction.

Operand restriction Specifying the same register for `<Rd>` and `<Rm>` was previously described as producing UNPREDICTABLE results. There is no restriction in ARMv6, and it is believed all relevant ARMv4 and ARMv5 implementations do not require this restriction either, because high performance multipliers read all their operands prior to writing back any results.

A4.1.41 MVN

31	28	27	26	25	24	23	22	21	20	19	16	15	12	11	0
cond		0	0	I	1	1	1	1	S	SBZ		Rd		shifter_operand	

`MVN` (Move Negative) generates the logical one's complement of a value. The value can be either an immediate value or a value from a register, and can be shifted before the MVN operation.

`MVN` can optionally update the condition code flags, based on the result.

Syntax

`MVN{<cond>}{S}  <Rd>, <shifter_operand>`

where:

`<cond>`   Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-7. If `<cond>` is omitted, the `AL` (always) condition is used.

`S`   Sets the S bit (bit[20]) in the instruction to 1 and specifies that the instruction updates the CPSR. If `S` is omitted, the S bit is set to 0 and the CPSR is not changed by the instruction. Two types of CPSR update can occur when `S` is specified:

   •   If `<Rd>` is not R15, the N and Z flags are set according to the result of the operation, and the C flag is set to the carry output bit generated by the shifter (see *Addressing Mode 1 - Data-processing operands* on page A5-2). The V flag and the rest of the CPSR are unaffected.

   •   If `<Rd>` is R15, the SPSR of the current mode is copied to the CPSR. This form of the instruction is UNPREDICTABLE if executed in User mode or System mode, because these modes do not have an SPSR.

`<Rd>`   Specifies the destination register.

`<shifter_operand>`

   Specifies the operand. The options for this operand are described in *Addressing Mode 1 - Data-processing operands* on page A5-2, including how each option causes the I bit (bit[25]) and the shifter_operand bits (bits[11:0]) to be set in the instruction.

   If the I bit is 0 and both bit[7] and bit[4] of shifter_operand are 1, the instruction is not `MVN`. Instead, see *Extending the instruction set* on page A3-35 to determine which instruction it is.

Architecture version

All.

Exceptions

None.

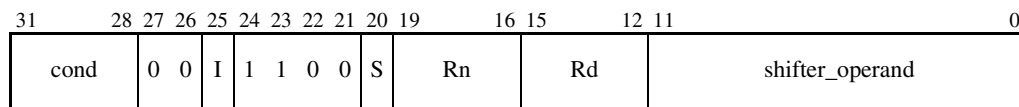### Operation

```
if ConditionPassed(cond) then
    Rd = NOT shifter_operand
    if S == 1 and Rd == R15 then
        if CurrentModeHasSPSR() then
            CPSR = SPSR
        else UNPREDICTABLE
    else if S == 1 then
        N Flag = Rd[31]
        Z Flag = if Rd == 0 then 1 else 0
        C Flag = shifter_carry_out
        V Flag = unaffected
```

Usage

Use MVN to:

- write a negative value into a register
- form a bit mask
- take the one's complement of a value.

ARM DDI 0100F-02

A4.1.42 ORR

31		28	27	26	25	24	23	22	21	20	19		16	15		12	11		0
cond			0	0	I	1	1	0	0	S	Rn			Rd			shifter_operand		

ORR (Logical OR) performs a bitwise (inclusive) OR of two values. The first value comes from a register. The second value can be either an immediate value or a value from a register, and can be shifted before the OR operation.

ORR can optionally update the condition code flags, based on the result.

Syntax

ORR{<cond>}{S} <Rd>, <Rn>, <shifter_operand>

where:

<cond>   Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-7. If <cond> is omitted, the AL (always) condition is used.

S   Sets the S bit (bit[20]) in the instruction to 1 and specifies that the instruction updates the CPSR. If S is omitted, the S bit is set to 0 and the CPSR is not changed by the instruction. Two types of CPSR update can occur when S is specified:

- If <Rd> is not R15, the N and Z flags are set according to the result of the operation, and the C flag is set to the carry output bit generated by the shifter (see *Addressing Mode 1 - Data-processing operands* on page A5-2). The V flag and the rest of the CPSR are unaffected.

- If <Rd> is R15, the SPSR of the current mode is copied to the CPSR. This form of the instruction is UNPREDICTABLE if executed in User mode or System mode, because these modes do not have an SPSR.

<Rd>   Specifies the destination register.

<Rn>   Specifies the register that contains the first operand.

<shifter_operand>

Specifies the second operand. The options for this operand are described in *Addressing Mode 1 - Data-processing operands* on page A5-2, including how each option causes the I bit (bit[25]) and the shifter_operand bits (bits[11:0]) to be set in the instruction.

If the I bit is 0 and both bit[7] and bit[4] of shifter_operand are 1, the instruction is not ORR. Instead, see *Extending the instruction set* on page A3-35 to determine which instruction it is.

Architecture version

All.

Exceptions

None.

Operation

```
if ConditionPassed(cond) then
    Rd = Rn OR shifter_operand
    if S == 1 and Rd == R15 then
        if CurrentModeHasSPSR() then
            CPSR = SPSR
        else UNPREDICTABLE
    else if S == 1 then
        N Flag = Rd[31]
        Z Flag = if Rd == 0 then 1 else 0
        C Flag = shifter_carry_out
        V Flag = unaffected
```

Usage

Use `ORR` to set selected bits in a register. For each bit, OR with 1 sets the bit, and OR with 0 leaves it unchanged.

A4.1.43 PKHBT

31	28	27 26 25 24 23 22 21 20	19	16	15	12	11	7	6	4	3	0
cond		0 1 1 0 1 0 0 0	Rn		Rd		shift_imm		0 0 1		Rm	

PKHBT (Pack Halfword Bottom Top) combines the bottom (least significant) halfword of its first operand with the top (most significant) halfword of its shifted second operand. The shift is a left shift, by any amount from 0 to 31.

Syntax

```
PKHBT {<cond>} <Rd>, <Rn>, <Rm> {, LSL #<shift_imm>}
```

where:

<cond>	Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-7. If <cond> is omitted, the AL (always) condition is used.
<Rd>	Specifies the destination register.
<Rn>	Specifies the register that contains the first operand. Bits[15:0] of this operand become bits[15:0] of the result of the operation.
<Rm>	Specifies the register that contains the second operand. This is shifted left by the specified amount, then bits[31:16] of this operand become bits[31:16] of the result of the operation.
<shift_imm>	Specifies the amount by which <Rm> is to be shifted left. If the shift specifier is omitted, a left shift by 0 is used.TBD find usual wording & fix.

Architecture version

Version 6 and above.

Exceptions

None.

Operation

```
if ConditionPassed(cond) then
    Rd[15:0] = Rn[15:0]
    Rd[31:16] = (Rm Logical_Shift_Left shift_imm)[31:16]
```

Usage

To construct the word in Rd consisting of the top half of register Ra and the bottom half of register Rb as its most and least significant halfwords respectively, use:
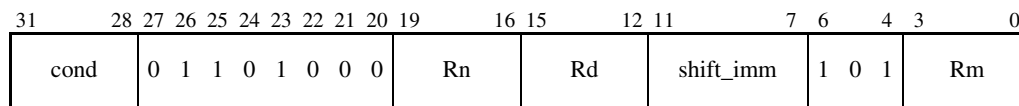
```
PKHBT   Rd, Rb, Ra
```

To construct the word in Rd consisting of the bottom half of register Ra and the bottom half of register Rb as its most and least significant halfwords respectively, use:
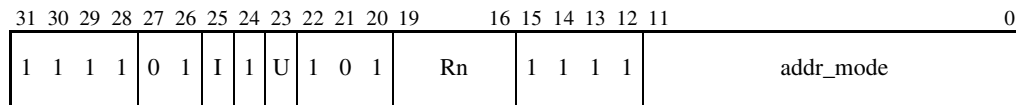
```
PKHBT   Rd, Rb, Ra, LSL #16
```

Notes

Use of R15   Specifying R15 for register `<Rd>`, `<Rm>`, or `<Rn>` has UNPREDICTABLE results.

A4.1.44 PKHTB

31		28	27	26	25	24	23	22	21	20	19	16	15	12	11	7	6	4	3	0
cond			0	1	1	0	1	0	0	0	Rn		Rd		shift_imm		1 0 1		Rm	

PKHTB (Pack Halfword Top Bottom) combines the top (most significant) halfword of its first operand with the bottom (least significant) halfword of its shifted second operand. The shift is an arithmetic right shift, by any amount from 1 to 32.

Syntax

```
PKHTB {<cond>} <Rd>, <Rn>, <Rm> {, ASR #<shift_imm>}
```

where:

<cond>	Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-7. If <cond> is omitted, the AL (always) condition is used.
<Rd>	Specifies the destination register.
<Rn>	Specifies the register that contains the first operand. Bits[31:16] of this operand become bits[31:16] of the result of the operation.
<Rm>	Specifies the register that contains the second operand. This is shifted right arithmetically by the specified amount, then bits[15:0] of this operand become bits[15:0] of the result of the operation.
<shift_imm>	Specifies the amount by which <Rm> is to be shifted right. A shift by 32 is encoded as shift_imm == 0.
	If the shift specifier is omitted, the assembler converts the instruction to PKHBT Rd, Rm, Rn. This produces the same effect as an arithmetic shift right by 0.

—— **Note** ——

If your assembler accepts shifts by #0 and treats them as equivalent to no shift or LSL #0, then it must accept ASR #0 here. It is equivalent to omitting the shift specifier.

Architecture version

Version 6 and above.

Exceptions

None.

Operation

```
if ConditionPassed(cond) then
    if shift_imm == 0 then          /* ASR #32 case */
        if Rm[31] == 0 then
            Rd[15:0] = 0x0000
        else
            Rd[15:0] = 0xFFFF
    else
        Rd[15:0] = (Rm Arithmetic_Shift_Right shift_imm)[15:0]
    Rd[31:16] = Rn[31:16]
```

Usage

To construct the word in Rd consisting of the top half of register Ra and the top half of register Rb as its most and least significant halfwords respectively, use:

```
    PKHTB   Rd, Ra, Rb, ASR #16
```

You can use this to truncate a Q31 number in Rb, and put the result into the bottom half of Rd. You can scale the Rb value by using a different shift amount.

To construct the word in Rd consisting of the top half of register Ra and the bottom half of register Rb as its most and least significant halfwords respectively, you can use:

```
    PKHTB   Rd, Ra, Rb
```

The assembler converts this into:

```
    PKHBT   Rd, Rb, Ra
```

Notes

Use of R15   Specifying R15 for register <Rd>, <Rm>, or <Rn> has UNPREDICTABLE results.

A4.1.45 PLD

31 30 29 28	27 26 25 24 23 22 21 20	19 16	15 14 13 12	11 0
1 1 1 1	0 1 I 1 U 1 0 1	Rn	1 1 1 1	addr_mode

PLD (Preload) signals the memory system that memory accesses from a specified address are likely in the near future. The memory system can respond by taking actions which 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 cache. PLD is a *hint* instruction, aimed at optimizing memory system performance. It has no architecturally defined effect, and memory systems that do not support this optimization can ignore it. On such memory systems, PLD acts as a NOP.

Syntax

```
PLD  <addressing_mode>
```

where:

```
<addressing_mode>
```

> Is described in *Addressing Mode 2 - Load and Store Word or Unsigned Byte* on page A5-19. It specifies the I, U, Rn, and addr_mode bits of the instruction. Only addressing modes with P == 1 and W == 0 are available for this instruction. Pre-indexed and post-indexed addressing modes have P == 0 or W == 1 and so are not available.

Architecture version

Version 6 and above, and version 5E, but not ARMv5TExP.

Exceptions

None.

Operation

```
/* No change occurs to programmer's model state, but where
 * appropriate, the memory system is signalled that memory accesses
 * to the specified address are likely in the near future.
 */
```

Notes

Condition   Unlike most other ARM instructions, PLD cannot be executed conditionally.

Write-back   Clearing bit[24] (the P bit) or setting bit[21] (the W bit) has UNPREDICTABLE results.

Data Aborts   This instruction never generates a Data Abort, nor does it signal any sort of memory system exception detected for the address generated by <addressing_mode> in any other way. All such memory system exceptions must be ignored by the memory system. Typically, the memory system does this by treating the PLD instruction as a NOP if any exceptional case is encountered while handling it.

Alignment   There are no alignment restrictions on the address generated by <addressing_mode>. If an implementation contains a System Control coprocessor (see Chapter B3 *The System Control Coprocessor*), it must not generate an alignment exception for any PLD instruction.

A4.1.46   QADD

31	28	27	26	25	24	23	22	21	20	19	16	15	12	11	8	7	6	5	4	3	0
cond		0	0	0	1	0	0	0	0	Rn		Rd		SBZ		0	1	0	1	Rm	
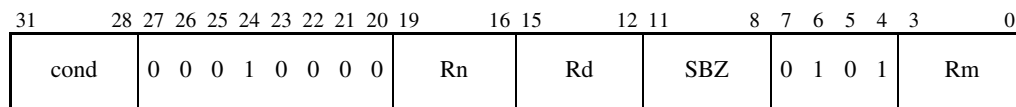
QADD (Saturating Add) performs integer addition.

It saturates the result to the 32-bit signed integer range $-2^{31} \leq x \leq 2^{31} - 1$.

If saturation occurs, QADD sets the Q flag in the CPSR.

Syntax

```
QADD{<cond>}  <Rd>, <Rm>, <Rn>
```

where:

<cond>   Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-7. If <cond> is omitted, the AL (always) condition is used.

<Rd>   Specifies the destination register.

<Rm>   Specifies the register that contains the first operand.

<Rn>   Specifies the register that contains the second operand.

Architecture version

Version 6 and above, and version 5E.

Exceptions

None.

Operation

```
if ConditionPassed(cond) then
    Rd = SignedSat(Rm + Rn, 32)
    if SignedDoesSat(Rm + Rn, 32) then
        Q Flag = 1
```

Usage

As well as performing saturated integer and Q31 additions, you can use `QADD` in combination with an `SMUL<x><y>`, `SMULW<y>`, or `SMULL` instruction to produce multiplications of Q15 and Q31 numbers. Three examples are:

* To multiply the Q15 numbers in the bottom halves of R0 and R1 and place the Q31 result in R2, use:

  ```
  SMULBB  R2, R0, R1
  QADD    R2, R2, R2
  ```

* To multiply the Q31 number in R0 by the Q15 number in the top half of R1 and place the Q31 result in R2, use:

  ```
  SMULWT  R2, R0, R1
  QADD    R2, R2, R2
  ```

* To multiply the Q31 numbers in R0 and R1 and place the Q31 result in R2, use:

  ```
  SMULL   R3, R2, R0, R1
  QADD    R2, R2, R2
  ```

Notes

Use of R15   Specifying R15 for register `<Rd>`, `<Rm>`, or `<Rn>` has UNPREDICTABLE results.

Condition flags   `QADD` does not affect the N, Z, C, or V flags.

A4.1.47   QADD16

31	28	27 26 25 24 23 22 21 20	19	16	15	12	11	8	7 6 5 4	3	0
cond		0 1 1 0 0 0 1 0	Rn		Rd		SBO		0 0 0 1	Rm	
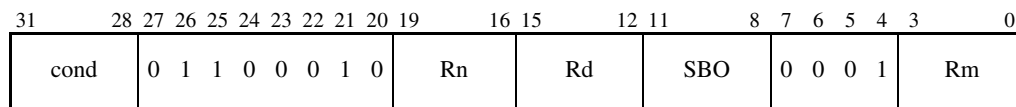
QADD16 performs two 16-bit integer additions.

It saturates the results to the 16-bit signed integer range $-2^{15} \leq x \leq 2^{15} - 1$.

QADD16 does not affect any flags.

Syntax

```
QADD16{<cond>}  <Rd>, <Rn>, <Rm>
```

where:

<cond>   Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-7. If <cond> is omitted, the AL (always) condition is used.

<Rd>   Specifies the destination register.

<Rn>   Specifies the register that contains the first operand.

<Rm>   Specifies the register that contains the second operand.

Architecture version

Version 6 and above.

Exceptions

None.

Operation

```
if ConditionPassed(cond) then
    Rd[15:0]  = SignedSat(Rn[15:0]  + Rm[15:0],  16)
    Rd[31:16] = SignedSat(Rn[31:16] + Rm[31:16], 16)
```

Usage

Use QADD16 in similar ways to the SADD16 instruction, but for signed saturated arithmetic. QADD16 does not set the GE bits for use with SEL. See *SADD16* on page A4-120 for more details.

Notes

Use of R15 Specifying R15 for register `<Rd>`, `<Rm>`, or `<Rn>` has UNPREDICTABLE results.

ARM DDI 0100F-02

A4.1.48 QADD8

31 28	27 26 25 24 23 22 21 20	19 16	15 12	11 8	7 6 5 4	3 0
cond	0 1 1 0 0 0 1 0	Rn	Rd	SBO	1 0 0 1	Rm
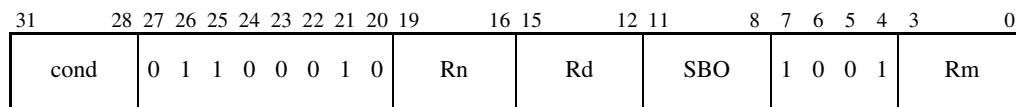
QADD8 performs four 8-bit integer additions.

It saturates the results to the 8-bit signed integer range $-2^7 \le x \le 2^7 - 1$.

QADD8 does not affect any flags.

Syntax

```
QADD8{<cond>}  <Rd>, <Rn>, <Rm>
```

where:

<cond>	Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-7. If <cond> is omitted, the AL (always) condition is used.
<Rd>	Specifies the destination register.
<Rn>	Specifies the register that contains the first operand.
<Rm>	Specifies the register that contains the second operand.

Architecture version

Version 6 and above.

Exceptions

None.

Operation

```
if ConditionPassed(cond) then
    Rd[7:0]   = SignedSat(Rn[7:0]   + Rm[7:0],    8)
    Rd[15:8]  = SignedSat(Rn[15:8]  + Rm[15:8],   8)
    Rd[23:16] = SignedSat(Rn[23:16] + Rm[23:16], 8)
    Rd[31:24] = SignedSat(Rn[31:24] + Rm[31:24], 8)
```

Usage

Use QADD8 in similar ways to the SADD8 instruction, but for signed saturated arithmetic. QADD8 does not set the GE bits for use with SEL. See *SADD8* on page A4-124 for more details.

Notes

Use of R15　　　　　Specifying R15 for register `<Rd>`, `<Rm>`, or `<Rn>` has UNPREDICTABLE results.

A4.1.49 QADDSUBX

31	28	27 26 25 24 23 22 21 20	19	16	15	12	11	8	7 6 5 4	3	0
cond		0 1 1 0 0 0 1 0	Rn		Rd		SBO		0 0 1 1	Rm	

QADDSUBX (Saturating Add and Subtract with Exchange) performs one 16-bit integer addition and one 16-bit subtraction.

It saturates the results to the 16-bit signed integer range $-2^{15} \le x \le 2^{15} - 1$.

It exchanges the two halfwords of the second operand before it performs the arithmetic.

QADDSUBX does not affect any flags.

Syntax

```
QADDSUBX{<cond>}  <Rd>, <Rn>, <Rm>
```

where:

<cond>   Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-7. If <cond> is omitted, the AL (always) condition is used.

<Rd>   Specifies the destination register.

<Rn>   Specifies the register that contains the first operand.

<Rm>   Specifies the register that contains the second operand.

Architecture version

Version 6 and above.

Exceptions

None.

Operation

```
if ConditionPassed(cond) then
    Rd[15:0]  = SignedSat(Rn[15:0]  - Rm[31:16], 16)
    Rd[31:16] = SignedSat(Rn[31:16] + Rm[15:0],  16)
```

Usage

You can use QADDSUBX for operations on complex numbers that are held as pairs of 16-bit integers or Q15 numbers. If you hold the real and imaginary parts of a complex number in the bottom and top half of a register respectively, then the instruction:

```
QADDSUBX  Rd, Ra, Rb
```

performs the complex arithmetic operation Rd = (Ra + i * Rb).

QADDSUBX does not set the Q flag, even if saturation occurs on either operation.

Notes

Use of R15   Specifying R15 for register <Rd>, <Rm>, or <Rn> has UNPREDICTABLE results.

A4.1.50 QDADD

31	28	27 26 25 24 23 22 21 20	19	16	15	12	11	8	7 6 5 4	3	0
cond		0 0 0 1 0 1 0 0	Rn		Rd		SBZ		0 1 0 1	Rm	
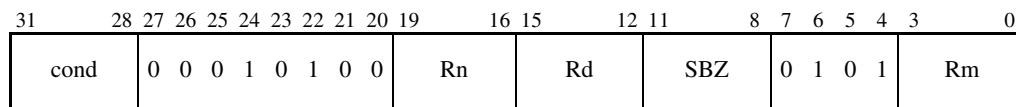
QDADD (Saturating Double and Add) doubles its second operand, then adds the result to its first operand.

Both the doubling and the addition have their results saturated to the 32-bit signed integer range $-2^{31} \leq x \leq 2^{31} - 1$.

If saturation occurs in either operation, the instruction sets the Q flag in the CPSR.

Syntax

```
QDADD{<cond>}  <Rd>, <Rm>, <Rn>
```

where:

<cond>   Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-7. If <cond> is omitted, the AL (always) condition is used.

<Rd>   Specifies the destination register.

<Rm>   Specifies the register that contains the first operand.

<Rn>   Specifies the register whose value is to be doubled, saturated, and used as the second operand for the saturated addition.

Rm and Rn are in reversed order in the assembler syntax, compared with the majority of ARM instructions.

Architecture version

Version 6 and above, and version 5E.

Exceptions

None.

Operation

```
if ConditionPassed(cond) then
    Rd = SignedSat(Rm + SignedSat(Rn*2, 32), 32)
    if SignedDoesSat(Rm + SignedSat(Rn*2, 32), 32) or
       SignedDoesSat(Rn*2, 32) then
        Q Flag = 1
```

Usage

The primary use for this instruction is to generate multiply-accumulate operations on Q15 and Q31 numbers, by placing it after an integer multiply instruction. Three examples are:

- To multiply the Q15 numbers in the top halves of R4 and R5 and add the product to the Q31 number in R6, use:

  ```
  SMULTT  R0, R4, R5
  QDADD   R6, R6, R0
  ```

- To multiply the Q15 number in the bottom half of R2 by the Q31 number in R3 and add the product to the Q31 number in R7, use:

  ```
  SMULWB  R0, R3, R2
  QDADD   R7, R7, R0
  ```

- To multiply the Q31 numbers in R2 and R3 and add the product to the Q31 number in R4, use:

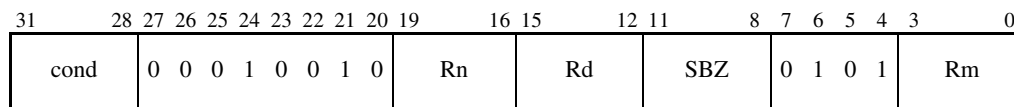 ```
  SMULL   R0, R1, R2, R3
  QDADD   R4, R4, R1
  ```

## Notes

**Use of R15**    Specifying R15 for register <Rd>, <Rm>, or <Rn> has UNPREDICTABLE results.

**Condition flags**   The QDADD instruction does not affect the N, Z, C, or V flags.

## A4.1.51 QDSUB

| 31 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 16 | 15 | 12 | 11 | 8 | 7 | 6 | 5 | 4 | 3 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| cond | | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | Rn | | Rd | | SBZ | | 0 | 1 | 0 | 1 | Rm | |
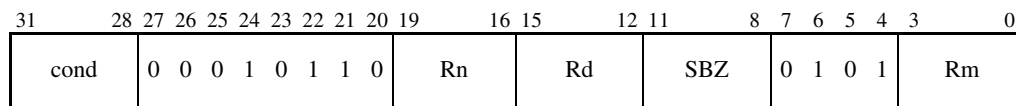
QDSUB (Saturating Double and Subtract) doubles its second operand, then subtracts the result from its first operand.

Both the doubling and the subtraction have their results saturated to the 32-bit signed integer range $-2^{31} \le x \le 2^{31} - 1$.

If saturation occurs in either operation, QDSUB sets the Q flag in the CPSR.

### Syntax

```
QDSUB{<cond>}  <Rd>, <Rm>, <Rn>
```

where:

<cond>   Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-7. If <cond> is omitted, the AL (always) condition is used.

<Rd>   Specifies the destination register.

<Rm>   Specifies the register that contains the first operand.

<Rn>   Specifies the register whose value is to be doubled, saturated, and used as the second operand for the saturated subtraction.

Rm and Rn are in reversed order in the assembler syntax, compared with the majority of ARM instructions.

Architecture version

Version 6 and above, and version 5E.

Exceptions

None.

Operation

```
if ConditionPassed(cond) then
    Rd = SignedSat(Rm - SignedSat(Rn*2, 32), 32)
    if SignedDoesSat(Rm - SignedSat(Rn*2, 32), 32) or
       SignedDoesSat(Rn*2, 32) then
       Q Flag = 1
```

Usage

The primary use for this instruction is to generate multiply-subtract operations on Q15 and Q31 numbers, by placing it after an integer multiply instruction. Three examples are:

*   To multiply the Q15 numbers in the top half of R4 and the bottom half of R5, and subtract the product from the Q31 number in R6, use:

    ```
    SMULTB  R0, R4, R5  QDSUB   R6, R6, R0
    ```

*   To multiply the Q15 number in the bottom half of R2 by the Q31 number in R3 and subtract the product from the Q31 number in R7, use:

    ```
    SMULWB  R0, R3, R2  QDSUB   R7, R7, R0
    ```

*   To multiply the Q31 numbers in R2 and R3 and subtract the product from the Q31 number in R4, use:

    ```
    SMULL   R0, R1, R2, R3  QDSUB   R4, R4, R1
    ```

Notes

Use of R15   Specifying R15 for register <Rd>, <Rm>, or <Rn> has UNPREDICTABLE results.

Condition flags   The QDSUB instruction does not affect the N, Z, C, or V flags.

A4.1.52 QSUB

31	28	27	26	25	24	23	22	21	20	19	16	15	12	11	8	7	6	5	4	3	0
cond		0	0	0	1	0	0	1	0	Rn		Rd		SBZ		0	1	0	1	Rm	

QSUB (Saturating Subtract) performs integer subtraction.

It saturates the result to the 32-bit signed integer range $-2^{31} \leq x \leq 2^{31} - 1$.

If saturation occurs, QSUB sets the Q flag in the CPSR.

Syntax

```
QSUB{<cond>}  <Rd>, <Rm>, <Rn>
```

where:

<cond>   Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-7. If <cond> is omitted, the AL (always) condition is used.

<Rd>   Specifies the destination register.

<Rm>   Specifies the register that contains the first operand.

<Rn>   Specifies the register that contains the second operand.

Rm and Rn are in reversed order in the assembler syntax, compared with the majority of ARM instructions.

Architecture version

Version 6 and above, and version 5E.

Exceptions

None.

Operation

```
if ConditionPassed(cond) then
    Rd = SignedSat(Rm - Rn, 32)
    if SignedDoesSat(Rm - Rn, 32) then
        Q Flag = 1
```

Notes

Use of R15   Specifying R15 for register <Rd>, <Rm>, or <Rn> has UNPREDICTABLE results.

Condition flags   QSUB does not affect the N, Z, C, or V flags.

Copyright © 1996–2003 ARM Limited. All rights reserved.
Confidential - Draft

A4.1.53 QSUB16

31	28	27 26 25 24 23 22 21 20	19	16	15	12	11	8	7 6 5 4	3	0
cond		0 1 1 0 0 0 1 0	Rn		Rd		SBO		0 1 1 1	Rm	

QSUB16 performs two 16-bit subtractions.

It saturates the results to the 16-bit signed integer range $-2^{15} \le x \le 2^{15} - 1$.

QSUB16 does not affect any flags.

Syntax

```
QSUB16{<cond>}  <Rd>, <Rn>, <Rm>
```

where:

<cond>   Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-7. If <cond> is omitted, the AL (always) condition is used.

<Rd>   Specifies the destination register.

<Rn>   Specifies the register that contains the first operand.

<Rm>   Specifies the register that contains the second operand.

Architecture version

Version 6 and above.

Exceptions

None.

Operation

```
if ConditionPassed(cond) then
    Rd[15:0]  = SignedSat(Rn[15:0]  - Rm[15:0],  16)
    Rd[31:16] = SignedSat(Rn[31:16] - Rm[31:16], 16)
```

Usage

Use QSUB16 in similar ways to the SSUB16 instruction, but for signed saturated arithmetic. QSUB16 does not set the GE bits for use with SEL. See *SSUB16* on page A4-188 for more details.

Notes

Use of R15   Specifying R15 for register `<Rd>`, `<Rm>`, or `<Rn>` has UNPREDICTABLE results.

A4.1.54 QSUB8

31	28	27 26 25 24 23 22 21 20	19	16	15	12	11	8	7 6 5 4	3	0
cond		0 1 1 0 0 0 1 0	Rn		Rd		SBO		1 1 1 1	Rm	

QSUB8 performs four 8-bit subtractions.

It saturates the results to the 8-bit signed integer range $-2^7 \leq x \leq 2^7 - 1$.

QSUB8 does not affect any flags.

Syntax

```
QSUB8{<cond>}  <Rd>, <Rn>, <Rm>
```

where:

<cond>   Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-7. If <cond> is omitted, the AL (always) condition is used.

<Rd>   Specifies the destination register.

<Rn>   Specifies the register that contains the first operand.

<Rm>   Specifies the register that contains the second operand.

Architecture version

Version 6 and above.

Exceptions

None.

Operation

```
if ConditionPassed(cond) then
    Rd[7:0]   = SignedSat(Rn[7:0]   - Rm[7:0],   8)
    Rd[15:8]  = SignedSat(Rn[15:8]  - Rm[15:8],  8)
    Rd[23:16] = SignedSat(Rn[23:16] - Rm[23:16], 8)
    Rd[31:24] = SignedSat(Rn[31:24] - Rm[31:24], 8)
```

Usage

Use QSUB8 in similar ways to SSUB8, but for signed saturated arithmetic. QSUB8 does not set the GE bits for use with SEL. See *SSUB8* on page A4-190 for more details.

Notes

Use of R15   Specifying R15 for register `<Rd>`, `<Rm>`, or `<Rn>` has UNPREDICTABLE results.

A4.1.55 QSUBADDX

31	28	27 26 25 24 23 22 21 20	19	16	15	12	11	8	7 6 5 4	3	0
cond		0 1 1 0 0 0 1 0	Rn		Rd		SBO		0 1 0 1	Rm	

QSUBADDX (Saturating Subtract and Add with Exchange) performs one 16-bit signed integer addition and one 16-bit signed integer subtraction, saturating the results to the 16-bit signed integer range $-2^{15} \leq x \leq 2^{15} - 1$. It exchanges the two halfwords of the second operand before it performs the arithmetic.

Syntax

```
QSUBADDX{<cond>}  <Rd>, <Rn>, <Rm>
```

where:

<cond>   Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-7. If <cond> is omitted, the AL (always) condition is used.

<Rd>   Specifies the destination register.

<Rn>   Specifies the register that contains the first operand.

<Rm>   Specifies the register that contains the second operand.

Architecture version

Version 6 and above.

Exceptions

None.

Operation

```
if ConditionPassed(cond) then
    Rd[15:0]  = SignedSat(Rn[15:0]  + Rm[31:16], 16)
    Rd[31:16] = SignedSat(Rn[31:16] - Rm[15:0],  16)
```

Usage

You can use QSUBADDX for operations on complex numbers that are held as pairs of 16-bit integers or Q15 numbers. If you hold the real and imaginary parts of a complex number in the bottom and top half of a register respectively, then the instruction:

```
QSUBADDX  Rd, Ra, Rb
```

performs the complex arithmetic operation Rd = (Ra – i * Rb).

QSUBADDX does not set the Q flag, even if saturation occurs on either operation.

Notes

Use of R15   Specifying R15 for register `<Rd>`, `<Rm>`, or `<Rn>` has UNPREDICTABLE results.

A4.1.56 REV

31	28 27					23 22	21 20	19	16 15	12 11	8	7 6	4 3	0
cond	0	1	1	0	1	0	1 1	SBO	Rd	SBO	0	0 1 1	Rm	
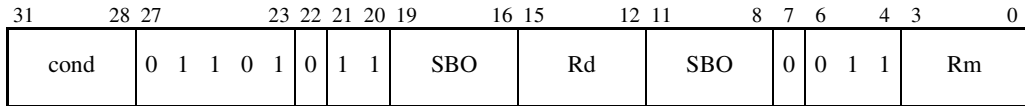
REV (Byte-Reverse Word) reverses the byte order in a 32-bit register.

Syntax

REV{<cond>} Rd, Rm

where:

<cond>   Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-7. If <cond> is omitted, the AL (always) condition is used.

<Rd>   Specifies the destination register.

<Rn>   Specifies the register that contains the operand.

Architecture version

Version 6 and above.

Exceptions

None.

Operation

```
if ConditionPassed(cond) then
    Rd[31:24] = Rm[ 7: 0]
    Rd[23:16] = Rm[15: 8]
    Rd[15: 8] = Rm[23:16]
    Rd[ 7: 0] = Rm[31:24]
```

Usage

Use REV to convert 32-bit big-endian data into little-endian data, or 32-bit little-endian data into big-endian data.

Notes

Use of R15   Specifying R15 for register <Rd> or <Rm> has UNPREDICTABLE results.

A4.1.57 REV16
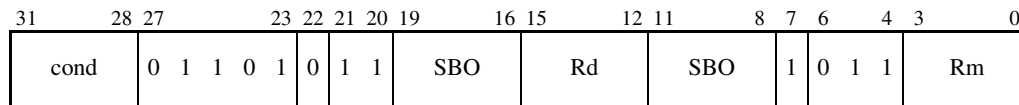
31	28	27				23	22	21	20	19		16	15		12	11		8	7	6		4	3		0
cond		0	1	1	0	1	0	1	1	SBO			Rd			SBO			1	0	1	1	Rm		

REV16 (Byte-Reverse Packed Halfword) reverses the byte order in each 16-bit halfword of a 32-bit register.

Syntax

```
REV16{<cond>} Rd, Rm
```

where:

<cond>   Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-7. If <cond> is omitted, the AL (always) condition is used.

<Rd>   Specifies the destination register.

<Rn>   Specifies the register that contains the operand.

Architecture version

Version 6 and above.

Exceptions

None.

Operation

```
if ConditionPassed(cond) then
    Rd[15: 8] = Rm[ 7: 0]
    Rd[ 7: 0] = Rm[15: 8]
    Rd[31:24] = Rm[23:16]
    Rd[23:16] = Rm[31:24]
```

Usage

Use REV16 to convert 16-bit big-endian data into little-endian data, or 16-bit little-endian data into big-endian data.

Notes

Use of R15   Specifying R15 for register <Rd> or <Rm> has UNPREDICTABLE results.

A4.1.58 REVSH

31	28	27				23	22	21	20	19		16	15		12	11		8	7	6		4	3		0

cond	0 1 1 0 1	1	1	1	SBO	Rd	SBO	1	0 1 1	Rm
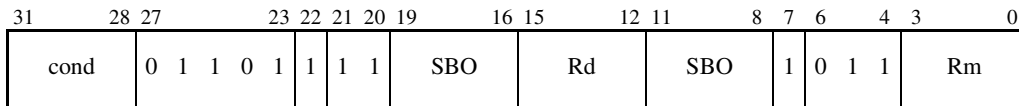
`REVSH` (Byte-Reverse Signed Halfword) reverses the byte order in the lower 16-bit halfword of a 32-bit register, and sign extends the result to 32-bits.

Syntax

```
REVSH{<cond>} Rd, Rm
```

where:

<cond>   Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-7. If <cond> is omitted, the AL (always) condition is used.

<Rd>   Specifies the destination register.

<Rn>   Specifies the register that contains the operand.

Architecture version

Version 6 and above.

Exceptions

None.

Operation

```
if ConditionPassed(cond) then
    Rd[15: 8] = Rm[ 7: 0]
    Rd[ 7: 0] = Rm[15: 8]
    if Rm[7] == 1 then
        Rd[31:16] = 0xFFFF
    else                          TBD Please check this carefully - it's not
        Rd[31:16] = 0x0000            what ARCH v6 SPEC 0002 E says!
```

Usage

Use `REVSH` to convert either:

- 16-bit signed big-endian data into 32-bit signed little-endian data
- 16-bit signed little-endian data into 32-bit signed big-endian data.

Notes

Use of R15   Specifying R15 for register `<Rd>` or `<Rm>` has UNPREDICTABLE results.

A4.1.59 RFE

31 30 29 28	27 26 25	24	23	22	21	20	19 16	15 12	11 10 9 8	7 0
1 1 1 1	1 0 0	P	U	0	W	1	Rn	SBZ	1 0 1 0	SBZ
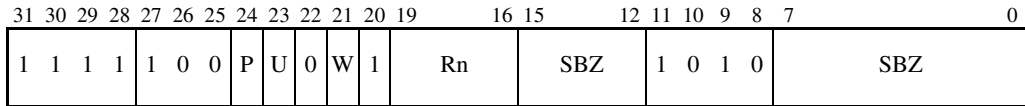
RFE (Return From Exception) loads the PC and the CPSR from the word at the specified address and the following word respectively.

Syntax

```
RFE<addressing_mode> <Rn>{!}
```

where:

`<addressing_mode>`

> Is similar to the `<addressing_mode>` in LDM and STM instructions, see *Addressing Mode 4 - Load and Store Multiple* on page A5-52, but with the following differences:
>
> • The number of registers to load is 2.
>
> • The register list is {PC, CPSR}.

`<Rn>` Specifies the base register to be used by `<addressing_mode>`. If R15 is specified as the base register, the result is UNPREDICTABLE.

`!` If present, sets the W bit. This causes the instruction to write a modified value back to its base register, in a manner similar to that specified for *Addressing Mode 4 - Load and Store Multiple* on page A5-52. If `!` is omitted, the W bit is 0 and the instruction does not change the base register.

Architecture version

Version 6 and above.

Exceptions

Data Abort.

Usage

TBD after operation...

Operation

```
address = start_address
value = Memory[address,4]
CPSR = Memory[address+4,4]
if (T Bit == 1) then
    PC = value AND 0xFFFFFFFE
else
    PC = value AND 0xFFFFFFFC
assert end_address == address + 8
```

where `start_address` and `end_address` are determined as described in *Addressing Mode 4 - Load and Store Multiple* on page A5-52, except that `Number_Of_Set_Bits_in(register_list)` evaluates to 2, rather than depending on bits[15:0] of the instruction.

Notes

Data Abort   For details of the effects of this instruction if a Data Abort occurs, see *Data Abort (data access memory abort)* on page A2-20.

Non word-aligned addresses

The behavior of `RFE` in relation to non word-aligned addresses in the base register is the same as the behavior of `LDM`.

Alignment   If an implementation includes a System Control coprocessor (see Chapter B3 *The System Control Coprocessor*), and alignment checking is enabled, an address with bits[1:0] != 0b00 causes an alignment exception.

Time order   The time order of the accesses to individual words of memory generated by `RFE` is not architecturally defined. Do not use this instruction on memory-mapped I/O locations where access order matters.

User mode   `RFE` obeys the usual rules about which bits of the CPSR can be written to in User mode, see *Types of PSR bits* on page A2-10 for details.

Condition   Unlike most other ARM instructions, `RFE` cannot be executed conditionally.

A4.1.60 RSB

31	28	27	26	25	24	23	22	21	20	19	16	15	12	11	0
cond		0	0	I	0	0	1	1	S	Rn		Rd		shifter_operand	
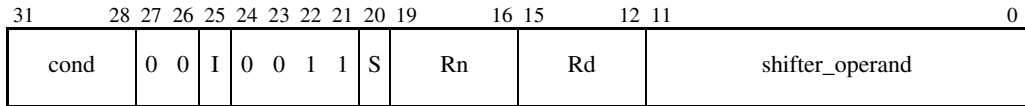
`RSB` (Reverse Subtract) subtracts a value from a second value.

The first value comes from a register. The second value can be either an immediate value or a value from a register, and can be shifted before the subtraction. This is the reverse of the normal order of operands in ARM assembler language.

`RSB` can optionally update the condition code flags, based on the result.

Syntax

```
RSB{<cond>}{S}  <Rd>, <Rn>, <shifter_operand>
```

where:

<cond>   Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-7. If `<cond>` is omitted, the `AL` (always) condition is used.

S   Sets the S bit (bit[20]) in the instruction to 1 and specifies that the instruction updates the CPSR. If `S` is omitted, the S bit is set to 0 and the CPSR is not changed by the instruction. Two types of CPSR update can occur when `S` is specified:

- If `<Rd>` is not R15, the N and Z flags are set according to the result of the subtraction, and the C and V flags are set according to whether the subtraction generated a borrow (unsigned underflow) and a signed overflow, respectively. The rest of the CPSR is unchanged.

- If `<Rd>` is R15, the SPSR of the current mode is copied to the CPSR. This form of the instruction is UNPREDICTABLE if executed in User mode or System mode, because these modes do not have an SPSR.

<Rd>   Specifies the destination register.

<Rn>   Specifies the register that contains the second operand.

<shifter_operand>

Specifies the first operand. The options for this operand are described in *Addressing Mode 1 - Data-processing operands* on page A5-2, including how each option causes the I bit (bit[25]) and the shifter_operand bits (bits[11:0]) to be set in the instruction.

If the I bit is 0 and both bit[7] and bit[4] of shifter_operand are 1, the instruction is not `RSB`. Instead, see *Extending the instruction set* on page A3-35 to determine which instruction it is.

Architecture version

All.

Exceptions

None.

Operation

```
if ConditionPassed(cond) then
    Rd = shifter_operand - Rn
    if S == 1 and Rd == R15 then
        if CurrentModeHasSPSR() then
            CPSR = SPSR
        else UNPREDICTABLE
    else if S == 1 then
        N Flag = Rd[31]
        Z Flag = if Rd == 0 then 1 else 0
        C Flag = NOT BorrowFrom(shifter_operand - Rn)
        V Flag = OverflowFrom(shifter_operand - Rn)
```

Usage

The following instruction stores the negation (two's complement) of `Rx` in `Rd`:

```
RSB Rd, Rx, #0
```

You can perform constant multiplication (of `Rx`) by $2^n - 1$ (into `Rd`) with:

```
RSB Rd, Rx, Rx, LSL #n
```

Notes

C flag   If `S` is specified, the C flag is set to:

       1   if no borrow occurs

       0   if a borrow does occur.

In other words, the C flag is used as a NOT(borrow) flag. This inversion of the borrow condition is usually compensated for by subsequent instructions. For example:

- The `SBC` and `RSC` instructions use the C flag as a NOT(borrow) operand, performing a normal subtraction if C == 1 and subtracting one more than usual if C == 0.

- The HS (unsigned higher or same) and LO (unsigned lower) conditions are equivalent to CS (carry set) and CC (carry clear) respectively.

    Copyright © 1996–2003 ARM Limited. All rights reserved.
Confidential - Draft

A4.1.61 RSC

31	28	27 26	25	24 23 22 21	20	19	16	15	12	11	0
cond		0 0	I	0 1 1 1	S	Rn		Rd		shifter_operand	

RSC (Reverse Subtract with Carry) subtracts one value from another, taking account of any borrow from a preceding less significant subtraction. The normal order of the operands is reversed, to allow subtraction from a shifted register value, or from an immediate value.

RSC can optionally update the condition code flags, based on the result.

Syntax

```
RSC{<cond>}{S}  <Rd>, <Rn>, <shifter_operand>
```

where:

<cond>   Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-7. If <cond> is omitted, the AL (always) condition is used.

S   Sets the S bit (bit[20]) in the instruction to 1 and specifies that the instruction updates the CPSR. If S is omitted, the S bit is set to 0 and the CPSR is not changed by the instruction. Two types of CPSR update can occur when S is specified:

                • If <Rd> is not R15, the N and Z flags are set according to the result of the subtraction, and the C and V flags are set according to whether the subtraction generated a borrow (unsigned underflow) and a signed overflow, respectively. The rest of the CPSR is unchanged.

                • If <Rd> is R15, the SPSR of the current mode is copied to the CPSR. This form of the instruction is UNPREDICTABLE if executed in User mode or System mode, because these modes do not have an SPSR.

<Rd>   Specifies the destination register.

<Rn>   Specifies the register that contains the second operand.

<shifter_operand>

                Specifies the first operand. The options for this operand are described in *Addressing Mode 1 - Data-processing operands* on page A5-2, including how each option causes the I bit (bit[25]) and the shifter_operand bits (bits[11:0]) to be set in the instruction.

                If the I bit is 0 and both bit[7] and bit[4] of shifter_operand are 1, the instruction is not RSC. Instead, see *Extending the instruction set* on page A3-35 to determine which instruction it is.

Architecture version

All.

Exceptions

None.

Operation

```
if ConditionPassed(cond) then
    Rd = shifter_operand - Rn - NOT(C Flag)
    if S == 1 and Rd == R15 then
        if CurrentModeHasSPSR() then
            CPSR = SPSR
        else UNPREDICTABLE
    else if S == 1 then
        N Flag = Rd[31]
        Z Flag = if Rd == 0 then 1 else 0
        C Flag = NOT BorrowFrom(shifter_operand - Rn - NOT(C Flag))
        V Flag = OverflowFrom(shifter_operand - Rn - NOT(C Flag))
```

Usage

Use RSC to synthesize multi-word subtraction, in cases where you need the order of the operands reversed to allow subtraction from a shifted register value, or from an immediate value.

Example

You can negate the 64-bit value in R0,R1 using the following sequence (R0 holds the least significant word), which stores the result in R2,R3:

```
    RSBS    R2,R0,#0
    RSC     R3,R1,#0
```

Notes

C flag   If S is specified, the C flag is set to:

1   if no borrow occurs

0   if a borrow does occur.

In other words, the C flag is used as a NOT(borrow) flag. This inversion of the borrow condition is usually compensated for by subsequent instructions. For example:

- The SBC and RSC instructions use the C flag as a NOT(borrow) operand, performing a normal subtraction if C == 1 and subtracting one more than usual if C == 0.

- The HS (unsigned higher or same) and LO (unsigned lower) conditions are equivalent to CS (carry set) and CC (carry clear) respectively.

A4.1.62 SADD16

31		28	27	26	25	24	23	22	21	20	19		16	15		12	11		8	7	6	5	4	3		0
cond			0	1	1	0	0	0	0	1		Rn			Rd			SBO		0	0	0	1		Rm	

SADD16 (Signed Add) performs two 16-bit signed integer additions. It sets the GE bits in the CPSR according to the results of the additions.

Syntax

```
SADD16{<cond>}  <Rd>, <Rn>, <Rm>
```

where:

<cond>   Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-7. If <cond> is omitted, the AL (always) condition is used.

<Rd>   Specifies the destination register.

<Rn>   Specifies the register that contains the first operand.

<Rm>   Specifies the register that contains the second operand.

Architecture version

Version 6 and above.

Exceptions

None.

Operation

```
if ConditionPassed(cond) then
    sum       = Rn[15:0] + Rm[15:0] /* Signed addition */
    Rd[15:0]  = sum[15:0]
    GE[1:0]   = if sum >= 0 then 0b11 else 0
    sum       = Rn[31:16] + Rm[31:16] /* Signed addition */
    Rd[31:16] = sum[15:0]
    GE[3:2]   = if sum >= 0 then 0b11 else 0
```

Usage

Use the SADD16 instruction to speed up operations on arrays of halfword data. For example, consider the instruction sequence:

```
LDR     R3, [R0], #4
LDR     R5, [R1], #4
SADD16  R3, R3, R4
STR     R3, [R2], #4
```

This performs the same operations as the instruction sequence:

```
LDRH    R3, [R0], #2
LDRH    R4, [R1], #2
ADD     R3, R3, R4
STRH    R3, [R2], #2
LDRH    R3, [R0], #2
LDRH    R4, [R1], #2
ADD     R3, R3, R4
STRH    R3, [R2], #2
```

The first sequence uses half as many instructions and typically half as many cycles as the second sequence.

You can also use SADD16 for operations on complex numbers that are held as pairs of 16-bit integers or Q15 numbers. If you hold the real and imaginary parts of a complex number in the bottom and top half of a register respectively, then the instruction:

```
SADD16  Rd, Ra, Rb
```

performs the complex arithmetic operation Rd = Ra + Rb.

SADD16 sets the GE flags according to the results of each addition. You can use these in a following SEL instruction. See *SEL* on page A4-134.

Notes

Use of R15   Specifying R15 for register <Rd>, <Rm>, or <Rn> has UNPREDICTABLE results.

A4.1.63 **SADD16TO32**

31	28	27	26	25	24	23	22	21	20	19	16	15	12	11	10	9	8	7	6	5	4	3	0
cond		0	1	1	0	1	0	1	1	Rn		Rd		rotate		SBZ		0	1	1	1	Rm	
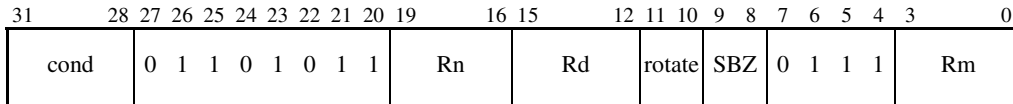
SADD16TO32 extracts a 16-bit value from a register, sign extends it to 32 bits, and adds the result to a value in another register. You can specify a rotation by 0, 8, 16, or 24 bits before extracting the 16-bit value.

Syntax

```
SADD16TO32{<cond>}  <Rd>, <Rn>, <Rm>{, <rotation>}
```

where:

`<cond>`	Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-7. If `<cond>` is omitted, the AL (always) condition is used.
`<Rd>`	Specifies the destination register.
`<Rn>`	Specifies the register that contains the first operand.
`<Rm>`	Specifies the register that contains the second operand.
`<rotation>`	This can be any one of:

- ROR #8. This is encoded as 0b01 in the rotate field.
- ROR #16. This is encoded as 0b10 in the rotate field.
- ROR #24. This is encoded as 0b11 in the rotate field.
- Omitted. This is encoded as 0b00 in the rotate field.

Note

If your assembler accepts shifts by #0 and treats them as equivalent to no shift or LSL #0, then it must accept ROR #0 here. It is equivalent to omitting `<rotation>`.

Architecture version

Version 6 and above.

Exceptions

None.

Operation

```
if ConditionPassed(cond) then
    operand2 = Rm Rotate_Right(8 * rotate)
    Rd       = Rn + SignExtend(operand2[15:0])
```

Usage

You can use `SADD16TO32` to eliminate a separate sign-extension instruction in many instruction sequences that act on **signed short** values in C/C++.
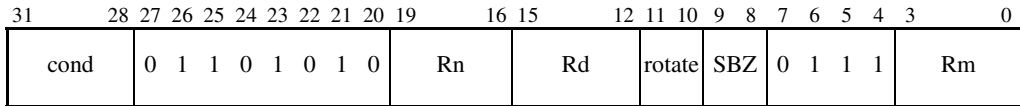
Notes

Use of R15   Specifying R15 for register `<Rd>` or `<Rm>` has UNPREDICTABLE results.

—— **Note** ——

Your assembler must fault the use of R15 for register `<Rn>`.

Encoding   If the `<Rn>` field of the instruction contains `0b1111`, the instruction is an `SUNPK16TO32` instruction instead, see *SUNPK16TO32* on page A4-216.

A4.1.64 SADD8

31		28	27 26 25 24 23 22 21 20	19	16	15	12	11	8	7 6 5 4	3	0
cond			0 1 1 0 0 0 0 1	Rn		Rd		SBO		1 0 0 1	Rm	

SADD8 performs four 8-bit signed integer additions. It sets the GE bits in the CPSR according to the results of the additions.

Syntax

```
SADD8{<cond>}  <Rd>, <Rn>, <Rm>
```

where:

<cond>   Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-7. If <cond> is omitted, the AL (always) condition is used.

<Rd>   Specifies the destination register.

<Rn>   Specifies the register that contains the first operand.

<Rm>   Specifies the register that contains the second operand.

Architecture version

Version 6 and above.

Exceptions

None.

Operation

```
if ConditionPassed(cond) then
    sum       = Rn[7:0] + Rm[7:0]      /* Signed addition */
    Rd[7:0]   = sum[7:0]
    GE[0]     = if sum >= 0 then 1 else 0
    sum       = Rn[15:8] + Rm[15:8]    /* Signed addition */
    Rd[15:8]  = sum[7:0]
    GE[1]     = if sum >= 0 then 1 else 0
    sum       = Rn[23:16] + Rm[23:16] /* Signed addition */
    Rd[23:16] = sum[7:0]
    GE[2]     = if sum >= 0 then 1 else 0
    sum       = Rn[31:24] + Rm[31:24] /* Signed addition */
    Rd[31:24] = sum[7:0]
    GE[3]     = if sum >= 0 then 1 else 0
```

Usage

Use SADD8 to speed up operations on arrays of byte data. This is similar to the way you can use the SADD16 instruction. See the usage subsection for *SADD16* on page A4-120 for details.

SADD8 sets the GE flags according to the results of each addition. You can use these in a following SEL instruction, see *SEL* on page A4-134.

Notes

Use of R15   Specifying R15 for register <Rd>, <Rm>, or <Rn> has UNPREDICTABLE results.

Confidential - Draft

A4.1.65 **SADD8TO16**

31	28	27	26	25	24	23	22	21	20	19	16	15	12	11	10	9	8	7	6	5	4	3	0
cond		0	1	1	0	1	0	0	0	Rn		Rd		rotate		SBZ		0	1	1	1	Rm	
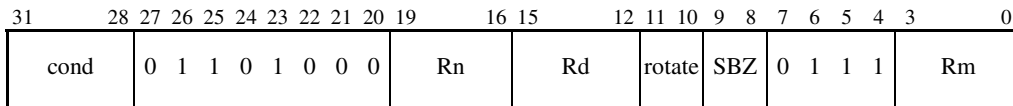
SADD8TO16 extracts two 8-bit values from a register, sign extends them to 16 bits each, and adds the results to two 16-bit values from another register. You can specify a rotation by 0, 8, 16, or 24 bits before extracting the 8-bit values.

Syntax

SADD8TO16{<cond>} <Rd>, <Rn>, <Rm>{, <rotation>}

where:

<cond>	Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-7. If <cond> is omitted, the AL (always) condition is used.
<Rd>	Specifies the destination register.
<Rn>	Specifies the register that contains the first operand.
<Rm>	Specifies the register that contains the second operand.
<rotation>	This can be any one of:

- ROR #8. This is encoded as 0b01 in the rotate field.
- ROR #16. This is encoded as 0b10 in the rotate field.
- ROR #24. This is encoded as 0b11 in the rotate field.
- Omitted. This is encoded as 0b00 in the rotate field.

——— **Note** ———

If your assembler accepts shifts by #0 and treats them as equivalent to no shift or LSL #0, then it must accept ROR #0 here. It is equivalent to omitting <rotation>.

Architecture version

Version 6 and above.

Exceptions

None.

Operation

```
if ConditionPassed(cond) then
    operand2  = Rm Rotate_Right(8 * rotate)
    Rd[15:0]  = Rn[15:0]  + SignExtend(operand2[7:0])
    Rd[31:16] = Rn[31:16] + SignExtend(operand2[23:16])
```

Usage

Use SADD8TO16 when you need to keep intermediate values to higher precision while working on arrays of signed byte values. See *UADD8TO16* on page A4-238 for an example of a similar usage.

Notes

Use of R15   Specifying R15 for register <Rd> or <Rm> has UNPREDICTABLE results.

——— **Note** ———

Your assembler must fault the use of R15 for register <Rn>.

Encoding   If the <Rn> field of the instruction contains 0b1111, the instruction is an SUNPK8TO16 instruction instead, see *SUNPK8TO16* on page A4-218.

A4.1.66 SADD8TO32

31	28	27	26	25	24	23	22	21	20	19	16	15	12	11	10	9	8	7	6	5	4	3	0
cond		0	1	1	0	1	0	1	0	Rn		Rd		rotate		SBZ		0	1	1	1	Rm	

SADD8TO32 extracts an 8-bit value from a register, sign extends it to 32 bits, and adds the result to the value in another register. You can specify a rotation by 0, 8, 16, or 24 bits before extracting the 8-bit value.

Syntax

```
SADD8TO32{<cond>}  <Rd>, <Rn>, <Rm>{, <rotation>}
```

where:

<cond>   Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-7. If <cond> is omitted, the AL (always) condition is used.

<Rd>   Specifies the destination register.

<Rn>   Specifies the register that contains the first operand.

<Rm>   Specifies the register that contains the second operand.

<rotation>   This can be any one of:

- ROR #8. This is encoded as 0b01 in the rotate field.
- ROR #16. This is encoded as 0b10 in the rotate field.
- ROR #24. This is encoded as 0b11 in the rotate field.
- Omitted. This is encoded as 0b00 in the rotate field.

—— **Note** ——

If your assembler accepts shifts by #0 and treats them as equivalent to no shift or LSL #0, then it must accept ROR #0 here. It is equivalent to omitting <rotation>.

Architecture version

Version 6 and above.

Exceptions

None.

ARM DDI 0100F-02

Operation

```
if ConditionPassed(cond) then
    operand2  = Rm Rotate_Right(8 * rotate)
    Rd = Rn + SignExtend(operand2[7:0])
```

Usage

You can use SADD8TO32 to eliminate a separate sign-extension instruction in many instruction sequences that act on **signed char** values in C/C++.

Notes

Use of R15   Specifying R15 for register <Rd> or <Rm> has UNPREDICTABLE results.

——— **Note** ———

Your assembler must fault the use of R15 for register <Rn>.

Encoding   If the <Rn> field of the instruction contains 0b1111, the instruction is an SUNPK8TO32 instruction instead, see *SUNPK8TO32* on page A4-220.

Copyright © 1996–2003 ARM Limited. All rights reserved.
Confidential - Draft

A4.1.67 SADDSUBX

31	28	27 26 25 24 23 22 21 20	19	16 15	12 11	8	7 6 5 4	3	0
cond		0 1 1 0 0 0 0 1	Rn	Rd	SBO		0 0 1 1	Rm	

SADDSUBX (Signed Add and Subtract with Exchange) performs one 16-bit signed integer addition and one 16-bit signed integer subtraction. It exchanges the two halfwords of the second operand before it performs the arithmetic. It sets the GE bits in the CPSR according to the results of the additions.

Syntax

```
SADDSUBX{<cond>}  <Rd>, <Rn>, <Rm>
```

where:

<cond>   Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-7. If <cond> is omitted, the AL (always) condition is used.

<Rd>   Specifies the destination register.

<Rn>   Specifies the register that contains the first operand.

<Rm>   Specifies the register that contains the second operand.

Architecture version

Version 6 and above.

Exceptions

None.

Operation

```
if ConditionPassed(cond) then
    diff      = Rn[15:0]  - Rm[31:16]    /* Signed subtraction */
    Rd[15:0]  = diff[15:0]
    GE[1:0]   = if diff >= 0 then 0b11 else 0
    sum       = Rn[31:16] + Rm[15:0]     /* Signed addition */
    Rd[31:16] = sum[15:0]
    GE[3:2]   = if sum >= 0 then 0b11 else 0
```

Usage

You can use `SADDSUBX` for operations on complex numbers that are held as pairs of 16-bit integers or Q15 numbers. If you hold the real and imaginary parts of a complex number in the bottom and top half of a register respectively, then the instruction:

```
SADDSUBX  Rd, Ra, Rb
```

performs the complex arithmetic operation Rd = Ra + (i * Rb).

`SADDSUBX` sets the GE flags according to the results the operation. You can use these in a following `SEL` instruction, see *SEL* on page A4-134.

Notes

Use of R15   Specifying R15 for register `<Rd>`, `<Rm>`, or `<Rn>` has UNPREDICTABLE results.

A4.1.68 SBC
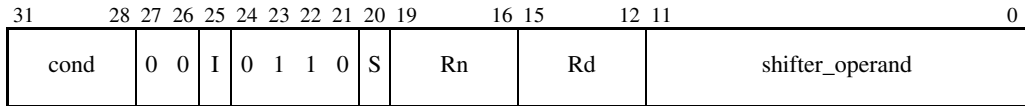
31	28	27 26	25	24 23 22 21	20	19	16	15	12	11	0
cond		0 0	I	0 1 1 0	S	Rn		Rd		shifter_operand	

SBC (Subtract with Carry) subtracts the value of its second operand and the value of NOT(Carry flag) from the value of its first operand.

The first operand comes from a register. The second operand can be either an immediate value or a value from a register, and can be shifted before the subtraction.

Use SBC to synthesize multi-word subtraction.

SBC can optionally update the condition code flags, based on the result.

Syntax

SBC{<cond>}{S} <Rd>, <Rn>, <shifter_operand>

where:

<cond>   Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-7. If <cond> is omitted, the AL (always) condition is used.

S   Sets the S bit (bit[20]) in the instruction to 1 and specifies that the instruction updates the CPSR. If S is omitted, the S bit is set to 0 and the CPSR is not changed by the instruction. Two types of CPSR update can occur when S is specified:

- If <Rd> is not R15, the N and Z flags are set according to the result of the subtraction, and the C and V flags are set according to whether the subtraction generated a borrow (unsigned underflow) and a signed overflow, respectively. The rest of the CPSR is unchanged.

- If <Rd> is R15, the SPSR of the current mode is copied to the CPSR. This form of the instruction is UNPREDICTABLE if executed in User mode or System mode, because these modes do not have an SPSR.

<Rd>   Specifies the destination register.

<Rn>   Specifies the register that contains the first operand.

<shifter_operand>

Specifies the second operand. The options for this operand are described in *Addressing Mode 1 - Data-processing operands* on page A5-2, including how each option causes the I bit (bit[25]) and the shifter_operand bits (bits[11:0]) to be set in the instruction.

If the I bit is 0 and both bit[7] and bit[4] of shifter_operand are 1, the instruction is not SBC. Instead, see *Extending the instruction set* on page A3-35 to determine which instruction it is.

Architecture version

All.

Exceptions

None.

Operation

```
if ConditionPassed(cond) then
    Rd = Rn - shifter_operand - NOT(C Flag)
    if S == 1 and Rd == R15 then
        if CurrentModeHasSPSR() then
            CPSR = SPSR
        else UNPREDICTABLE
    else if S == 1 then
        N Flag = Rd[31]
        Z Flag = if Rd == 0 then 1 else 0
        C Flag = NOT BorrowFrom(Rn - shifter_operand - NOT(C Flag))
        V Flag = OverflowFrom(Rn - shifter_operand - NOT(C Flag))
```

Usage

If register pairs R0,R1 and R2,R3 hold 64-bit values (R0 and R2 hold the least significant words), the following instructions leave the 64-bit difference in R4,R5:

```
SUBS    R4,R0,R2
SBC     R5,R1,R3
```

Notes

C flag If `S` is specified, the C flag is set to:

1   if no borrow occurs

0   if a borrow does occur.

In other words, the C flag is used as a NOT(borrow) flag. This inversion of the borrow condition is used by subsequent instructions: `SBC` and `RSC` use the C flag as a NOT(borrow) operand, performing a normal subtraction if C == 1 and subtracting one more than usual if C == 0.

The HS (unsigned higher or same) and LO (unsigned lower) conditions are equivalent to CS (carry set) and CC (carry clear) respectively.

A4.1.69 SEL

31	28	27 26 25 24 23 22 21 20	19	16	15	12	11	8	7 6 5 4	3	0
cond		0 1 1 0 1 0 0 0	Rn		Rd		SBO		1 0 1 1	Rm	

SEL (Select) selects each byte of its result from either its first operand or its second operand, according to the values of the GE flags.

Syntax

```
SEL{<cond>}  <Rd>, <Rn>, <Rm>
```

where:

<cond>   Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-7. If <cond> is omitted, the AL (always) condition is used.

<Rd>   Specifies the destination register.

<Rn>   Specifies the register that contains the first operand.

<Rm>   Specifies the register that contains the second operand.

Architecture version

Version 6 and above.

Exceptions

None.

Operation

```
if ConditionPassed(cond) then
    Rd[7:0]   = if GE[0] == 1 then Rn[7:0]   else Rm[7:0]
    Rd[15:8]  = if GE[1] == 1 then Rn[15:8]  else Rm[15:8]
    Rd[23:16] = if GE[2] == 1 then Rn[23:16] else Rm[23:16]
    Rd[31:24] = if GE[3] == 1 then Rn[31:24] else Rm[31:24]
```

Usage

Use SEL after instructions such as SADD, SSUB, UADD, and USUB, that set the GE flags. For example, the following sequence of instructions sets each byte of Rd equal to the unsigned minimum of the corresponding bytes of Ra and Rb:

```
    USUB8  Rd, Ra, Rb
    SEL    Rd, Rb, Ra
```

Notes

Use of R15   Specifying R15 for register `<Rd>`, `<Rm>`, or `<Rn>` has UNPREDICTABLE results.

A4.1.70 SETEND

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15			10	9	8	7		4	3			0
1	1	1	1	0	0	0	1	0	0	0	0	0	0	0	1	SBZ				E	SBZ	0	0	0	0	SBZ		

SETEND modifies the CPSR E bit, without changing any other bits in the CPSR.

Syntax

```
SETEND <endian_specifier>
```

TBD reword this section!

where:

```
<endian_specifier>
```

    Is one of:

    BE   Sets the E bit in the instruction. This sets the CPSR E bit.

    LE   Clears the E bit in the instruction. This clears the CPSR E bit.

Architecture version

Version 6 and above.

Exceptions

None.

Operation

```
CPSR = CPSR with specified E bit modification
```

Usage

Use SETEND to change the byte order for data accesses. You can use SETEND to increase the efficiency of access to a series of big-endian data fields in an otherwise little-endian application, or to a series of little-endian data fields in an otherwise big-endian application.

Notes

Condition Unlike most other ARM instructions, SETEND cannot be executed conditionally.

    ARM DDI 0100F-02

A4.1.71 SHADD16

31	28	27 26 25 24 23 22 21 20	19	16	15	12	11	8	7 6 5 4	3	0
cond		0 1 1 0 0 0 1 1	Rn		Rd		SBO		0 0 0 1	Rm	

SHADD16 (Signed Halving Add) performs two 16-bit signed integer additions, and halves the results.

SHADD16 has no effect on the GE flags.

Syntax

```
SHADD16{<cond>}  <Rd>, <Rn>, <Rm>
```

where:

<cond>   Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-7. If <cond> is omitted, the AL (always) condition is used.

<Rd>   Specifies the destination register.

<Rn>   Specifies the register that contains the first operand.

<Rm>   Specifies the register that contains the second operand.

Architecture version

Version 6 and above.

Exceptions

None.

Operation

```
if ConditionPassed(cond) then
    sum       = Rn[15:0] + Rm[15:0] /* Signed addition */
    Rd[15:0]  = sum[16:1]
    sum       = Rn[31:16] + Rm[31:16] /* Signed addition */
    Rd[31:16] = sum[16:1]
```

Usage

Use SHADD16 for similar purposes to SADD16 (see *SADD16* on page A4-120). SHADD16 averages the operands.

SHADD16 does not set any flags, as overflow is not possible.

Notes

Use of R15   Specifying R15 for register `<Rd>`, `<Rm>`, or `<Rn>` has UNPREDICTABLE results.

A4.1.72 SHADD8

31	28	27 26 25 24 23 22 21 20	19	16	15	12	11	8	7 6 5 4	3	0
cond		0 1 1 0 0 0 1 1	Rn		Rd		SBO		1 0 0 1	Rm	
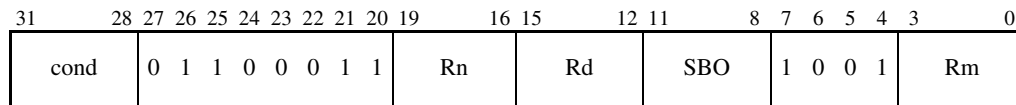
SHADD8 performs four 8-bit signed integer additions, and halves the results.

SHADD8 has no effect on the GE flags.

Syntax

```
SHADD8{<cond>}  <Rd>, <Rn>, <Rm>
```

where:

<cond>   Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-7. If <cond> is omitted, the AL (always) condition is used.

<Rd>   Specifies the destination register.

<Rn>   Specifies the register that contains the first operand.

<Rm>   Specifies the register that contains the second operand.

Architecture version

Version 6 and above.

Exceptions

None.

Operation

```
if ConditionPassed(cond) then
    sum       = Rn[7:0] + Rm[7:0] /* Signed addition */
    Rd[7:0]   = sum[8:1]
    sum       = Rn[15:8] + Rm[15:8] /* Signed addition */
    Rd[15:8]  = sum[8:1]
    sum       = Rn[23:16] + Rm[23:16] /* Signed addition */
    Rd[23:16] = sum[8:1]
    sum       = Rn[31:24] + Rm[31:24] /* Signed addition */
    Rd[31:24] = sum[8:1]
```

Usage

Use SHADD8 similar purposes to SADD16 (see *SADD16* on page A4-120). SHADD8 averages the operands.

Copyright © 1996–2003 ARM Limited. All rights reserved.
Confidential - Draft

`SHADD8` does not set any flags, as overflow is not possible.

Notes

Use of R15   Specifying R15 for register `<Rd>`, `<Rm>`, or `<Rn>` has UNPREDICTABLE results.

A4.1.73 SHADDSUBX

31	28	27	26	25	24	23	22	21	20	19	16	15	12	11	8	7	6	5	4	3	0
cond		0	1	1	0	0	0	1	1	Rn		Rd		SBO		0	0	1	1	Rm	
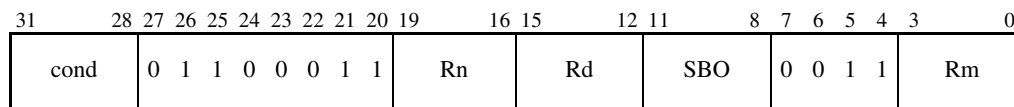
SHADDSUBX (Signed Halving Add and Subtract with Exchange) performs one 16-bit signed integer addition and one 16-bit signed integer subtraction, and halves the results. It exchanges the two halfwords of the second operand before it performs the arithmetic.

SHADDSUBX has no effect on the GE flags.

Syntax

```
SHADDSUBX{<cond>}  <Rd>, <Rn>, <Rm>
```

where:

<cond>   Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-7. If <cond> is omitted, the AL (always) condition is used.

<Rd>   Specifies the destination register.

<Rn>   Specifies the register that contains the first operand.

<Rm>   Specifies the register that contains the second operand.

Architecture version

Version 6 and above.

Exceptions

None.

Operation

```
if ConditionPassed(cond) then
    diff      = Rn[15:0]  - Rm[31:16]    /* Signed subtraction */
    Rd[15:0]  = diff[16:1]
    sum       = Rn[31:16] + Rm[15:0]     /* Signed addition */
    Rd[31:16] = sum[16:1]
```

Usage

Use SHADDSUBX for similar purposes to SADDSUBX, but when you want the results halved. See *SADDSUBX* on page A4-130 for further details.

SHADDSUBX does not set any flags, as overflow is not possible.

Notes

Use of R15   Specifying R15 for register `<Rd>`, `<Rm>`, or `<Rn>` has UNPREDICTABLE results.

ARM DDI 0100F-02

A4.1.74 SHSUB16

31	28	27 26 25 24 23 22 21 20	19	16	15	12	11	8	7 6 5 4	3	0
cond		0 1 1 0 0 0 1 1	Rn		Rd		SBO		0 1 1 1	Rm	
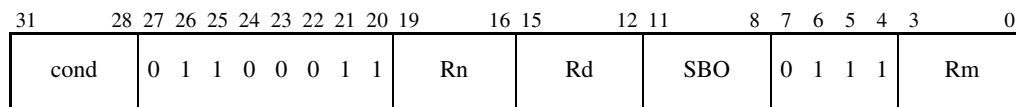
`SHSUB16` (Signed Halving Subtract) performs two 16-bit signed integer subtractions, and halves the results.

`SHSUB16` has no effect on the GE flags.

Syntax

```
SHSUB16{<cond>}  <Rd>, <Rn>, <Rm>
```

where:

<cond>   Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-7. If `<cond>` is omitted, the `AL` (always) condition is used.

<Rd>   Specifies the destination register.

<Rn>   Specifies the register that contains the first operand.

<Rm>   Specifies the register that contains the second operand.

Architecture version

Version 6 and above.

Exceptions

None.

Operation

```
if ConditionPassed(cond) then
    diff      = Rn[15:0] - Rm[15:0]    /* Signed subtraction */
    Rd[15:0]  = diff[16:1]
    diff      = Rn[31:16] - Rm[31:16]  /* Signed subtraction */
    Rd[31:16] = diff[16:1]
```

Usage

Use `SHSUB16` to speed up operations on arrays of halfword data. This is similar to the way you can use `SADD16`. See the usage subsection for *SADD16* on page A4-120 for details.

You can also use `SHSUB16` for operations on complex numbers that are held as pairs of 16-bit integers or Q15 numbers. If you hold the real and imaginary parts of a complex number in the bottom and top half of a register respectively, then the instruction:

```
SHSUB16  Rd, Ra, Rb
```

performs the complex arithmetic operation Rd = (Ra - Rb)/2.

`SHSUB16` does not set any flags, as overflow is not possible.

Notes

Use of R15   Specifying R15 for register `<Rd>`, `<Rm>`, or `<Rn>` has UNPREDICTABLE results.

A4.1.75 **SHSUB8**

31	28 27 26 25 24 23 22 21 20	19	16 15	12 11	8 7 6 5 4 3	0
cond	0 1 1 0 0 0 1 1	Rn	Rd	SBO	1 1 1 1	Rm
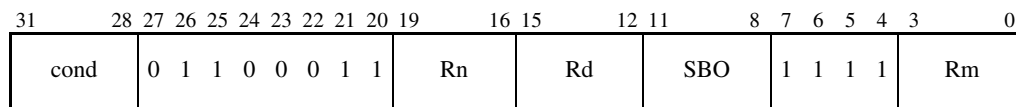
SHSUB8 performs four 8-bit signed integer subtractions, and halves the results.

SHSUB8 has no effect on the GE flags.

Syntax

```
SHSUB8{<cond>}  <Rd>, <Rn>, <Rm>
```

where:

<cond>   Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-7. If <cond> is omitted, the AL (always) condition is used.

<Rd>   Specifies the destination register.

<Rn>   Specifies the register that contains the first operand.

<Rm>   Specifies the register that contains the second operand.

Architecture version

Version 6 and above.

Exceptions

None.

Operation

```
if ConditionPassed(cond) then
    diff      = Rn[7:0] - Rm[7:0]      /* Signed subtraction */
    Rd[7:0]   = diff[8:1]
    diff      = Rn[15:8] - Rm[15:8]    /* Signed subtraction */
    Rd[15:8]  = diff[8:1]
    diff      = Rn[23:16] - Rm[23:16]  /* Signed subtraction */
    Rd[23:16] = diff[8:1]
    diff      = Rn[31:24] - Rm[31:24]  /* Signed subtraction */
    Rd[31:24] = diff[8:1]
```

Usage

Use SHSUB8 to speed up operations on arrays of byte data. This is similar to the way you can use SADD16 to speed up operations on halfword data. See the usage subsection for *SADD16* on page A4-120 for details.

SHSUB8 does not set any flags, as overflow is not possible.

Notes

Use of R15   Specifying R15 for register <Rd>, <Rm>, or <Rn> has UNPREDICTABLE results.

A4.1.76 SHSUBADDX

31	28	27 26 25 24 23 22 21 20	19	16	15	12	11	8	7 6 5 4	3	0
cond		0 1 1 0 0 0 1 1	Rn		Rd		SBO		0 1 0 1	Rm	
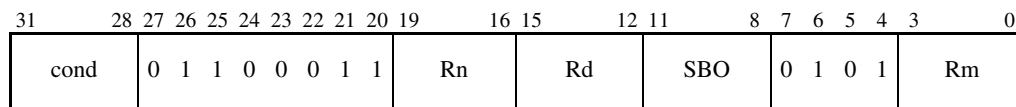
SHSUBADDX (Signed Halving Subtract and Add with Exchange) performs one 16-bit signed integer addition and one 16-bit signed integer subtraction, and halves the results. It exchanges the two halfwords of the second operand before it performs the arithmetic.

SHSUBADDX has no effect on the GE flags.

Syntax

```
SHSUBADDX{<cond>}   <Rd>, <Rn>, <Rm>
```

where:

<cond>   Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-7. If <cond> is omitted, the AL (always) condition is used.

<Rd>   Specifies the destination register.

<Rn>   Specifies the register that contains the first operand.

<Rm>   Specifies the register that contains the second operand.

Architecture version

Version 6 and above.

Exceptions

None.

Operation

```
if ConditionPassed(cond) then
    sum       = Rn[15:0]  + Rm[31:16]    /* Signed addition */
    Rd[15:0]  = sum[16:1]
    diff      = Rn[31:16] - Rm[15:0]     /* Signed subtraction */
    Rd[31:16] = diff[16:1]
```

Usage

Use SHSUBADDX for similar purposes to SSUBADDX, but when you want the results halved. See *SSUBADDX* on page A4-192 for further details.

SHSUBADDX does not set any flags, as overflow is not possible.

Notes

Use of R15   Specifying R15 for register <Rd>, <Rm>, or <Rn> has UNPREDICTABLE results.

A4.1.77 SMLA<x><y>

31	28	27 26 25 24 23 22 21 20	19	16	15	12	11	8	7	6	5	4	3	0
cond		0 0 0 1 0 0 0 0	Rd		Rn		Rs		1	y	x	0	Rm	

SMLA<x><y> (Signed multiply-accumulate BB, BT, TB, and TT) 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 CPSR. It is not possible for overflow to occur during the multiplication.

Syntax

SMLA<x><y>{<cond>} <Rd>, <Rm>, <Rs>, <Rn>

where:

<x>   Specifies which half of the source register <Rm> is used as the first multiply operand. If <x> is B, then x == 0 in the instruction encoding and the bottom half (bits[15:0]) of <Rm> is used. If <x> is T, then x == 1 in the instruction encoding and the top half (bits[31:16]) of <Rm> is used.

<y>   Specifies which half of the source register <Rs> is used as the second multiply operand. If <y> is B, then y == 0 in the instruction encoding and the bottom half (bits[15:0]) of <Rs> is used. If <y> is T, then y == 1 in the instruction encoding and the top half (bits[31:16]) of <Rs> is used.

<cond>   Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-7. If <cond> is omitted, the AL (always) condition is used.

<Rd>   Specifies the destination register.

<Rm>   Specifies the source register whose bottom or top half (selected by <x>) is the first multiply operand.

<Rs>   Specifies the source register whose bottom or top half (selected by <y>) is the second multiply operand.

<Rn>   Specifies the register which contains the accumulate value.

Architecture version

Version 6 and above, and version 5E.

Exceptions

None.

Operation

```
if ConditionPassed(cond) then

    if (x == 0) then
        operand1 = SignExtend(Rm[15:0])
    else /* x == 1 */
        operand1 = SignExtend(Rm[31:16])

    if (y == 0) then
        operand2 = SignExtend(Rs[15:0])
    else /* y == 1 */
        operand2 = SignExtend(Rs[31:16])

    Rd = (operand1 * operand2) + Rn
    if OverflowFrom((operand1 * operand2) + Rn) then
        Q Flag = 1
```

Usage

In addition to its straightforward uses for integer multiply-accumulates, these instructions sometimes provide a faster alternative to $Q15 \times Q15 + Q31 \rightarrow Q31$ multiply-accumulates synthesized from SMUL<x><y> and QDADD instructions. The main circumstances under which this is possible are:

- if it is known that saturation and/or overflow cannot occur during the calculation

- if saturation and/or overflow can occur during the calculation but the Q flag is going to be used to detect this and take remedial action if it does occur.

For example, the following code produces the dot product of the four Q15 numbers in R0 and R1 by the four Q15 numbers in R2 and R3:

```
SMULBB   R4, R0, R2
QADD     R4, R4, R4
SMULTT   R5, R0, R2
QDADD    R4, R4, R5
SMULBB   R5, R1, R3
QDADD    R4, R4, R5
SMULTT   R5, R1, R3
QDADD    R4, R4, R5
```

In the absence of saturation, the following code provides a faster alternative:

```
SMULBB   R4, R0, R2
SMLATT   R4, R0, R2, R4
SMLABB   R4, R1, R3, R4
SMLATT   R4, R1, R3, R4
QADD     R4, R4, R4
```

Furthermore, if saturation and/or overflow occurs in this second sequence, it sets the Q flag. This allows remedial action to be taken, such as scaling down the data values and repeating the calculation.

Notes

Use of R15   Specifying R15 for register <Rd>, <Rm>, <Rs>, or <Rn> has UNPREDICTABLE results.

Condition flags   The SMLA<x><y> instructions do not affect the N, Z, C, or V flags.

A4.1.78 SMLAD

31 28	27 26 25 24 23 22 21 20	19 16	15 12	11 8	7 6 5 4	3 0
cond	0 1 1 1 0 0 0 0	Rd	Rn	Rs	0 0 X 1	Rm
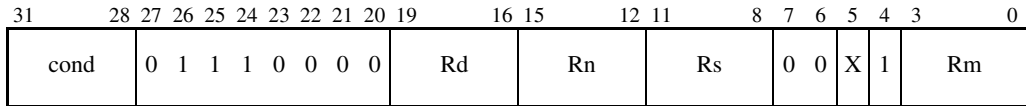
SMLAD (Signed Multiply Accumulate Dual) performs two signed 16 x 16-bit multiplications. It adds the products to a 32-bit accumulate operand.

Optionally, you can exchange the halfwords of the second operand before performing the arithmetic. This produces top ∞ bottom and bottom ∞ top multiplication.

This instruction sets the Q flag if the accumulate operation overflows. Overflow cannot occur during the multiplications.

Syntax

```
SMLAD{X}{<cond>}  <Rd>, <Rm>, <Rs>, <Rn>
```

where:

X	Sets the X bit of the instruction to 1, and the multiplications are bottom ∞ top and top ∞ bottom.
	If the X is omitted, sets the X bit to 0, and the multiplications are bottom ∞ bottom and top ∞ top.
<cond>	Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-7. If <cond> is omitted, the AL (always) condition is used.
<Rd>	Specifies the destination register.
<Rm>	Specifies the register that contains the first operand.
<Rs>	Specifies the register that contains the second operand.
<Rn>	Specifies the register that contains the accumulate operand.

Architecture version

Version 6 and above.

Exceptions

None.

Operation

```
if ConditionPassed(cond) then
    if X == 1 then
        operand2 = Rs Rotate_Right 16
    else
        operand2 = Rs
    product1 = Rm[15:0]  * operand2[15:0]     /* Signed multiplication */
    product2 = Rm[31:16] * operand2[31:16]    /* Signed multiplication */
    Rd = Rn + product1 + product2
    if OverflowFrom(Rn + product1 + product2) then
        Q flag = 1
```

Usage

Use SMLAD to accumulate the sums of products of 16-bit data, with a 32-bit accumulator. This instruction enables you to do this at approximately twice the speed otherwise possible. This is useful in many applications, for example in filters.

You can use the X option for calculating the imaginary part for similar filters acting on complex numbers with 16-bit real and 16-bit imaginary parts.

Notes

Use of R15   Specifying R15 for register <Rd>, <Rm>, or <Rs> has UNPREDICTABLE results.

——— **Note** ———

Your assembler must fault the use of R15 for register <Rn>.

Encoding   If the <Rn> field of the instruction contains 0b1111, the instruction is an SMUAD instruction instead, see *SMUAD* on page A4-172.

Early termination   If the multiplier implementation supports early termination, it must be implemented on the value of the <Rs> operand. The type of early termination used (signed or unsigned) is IMPLEMENTATION DEFINED.

C and V flags   The SMLAD instruction leaves the C and V flags unchanged.

A4.1.79 SMLAL

31	28	27 26 25 24 23 22 21	20	19	16	15	12	11	8	7 6 5 4	3	0
cond		0 0 0 0 1 1 1	S	RdHi		RdLo		Rs		1 0 0 1	Rm	

SMLAL (Signed Multiply Accumulate Long) multiplies two signed 32-bit values to produce a 64-bit value, and accumulates this with a 64-bit value.

SMLAL can optionally update the condition code flags, based on the result.

Syntax

SMLAL{<cond>}{S} <RdLo>, <RdHi>, <Rm>, <Rs>

where:

<cond>   Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-7. If <cond> is omitted, the AL (always) condition is used.

S   Causes the S bit (bit[20]) in the instruction to be set to 1 and specifies that the instruction updates the CPSR by setting the N and Z flags according to the result of the multiply-accumulate. If S is omitted, the S bit of the instruction is set to 0 and the entire CPSR is unaffected by the instruction.

<RdLo>   Supplies the lower 32 bits of the value to be added to the product of <Rm> and <Rs>, and is the destination register for the lower 32 bits of the result.

<RdHi>   Supplies the upper 32 bits of the value to be added to the product of <Rm> and <Rs>, and is the destination register for the upper 32 bits of the result.

<Rm>   Holds the signed value to be multiplied with the value of <Rs>.

<Rs>   Holds the signed value to be multiplied with the value of <Rm>.

Architecture version

Versions 3M and 4 and above, excluding xM versions.

Exceptions

None.

Operation

```
if ConditionPassed(cond) then
    RdLo = (Rm * Rs)[31:0] + RdLo /* Signed multiplication */
    RdHi = (Rm * Rs)[63:32] + RdHi + CarryFrom((Rm * Rs)[31:0] + RdLo)
    if S == 1 then
        N Flag = RdHi[31]
        Z Flag = if (RdHi == 0) and (RdLo == 0) then 1 else 0
        C Flag = unaffected    /* See "C and V flags" note */
        V Flag = unaffected    /* See "C and V flags" note */
```

Usage

SMLAL multiplies signed variables to produce a 64-bit result, which is added to the 64-bit value in the two destination general-purpose registers. The result is written back to the two destination general-purpose registers.

Notes

Use of R15 Specifying R15 for register <RdHi>, <RdLo>, <Rm>, or <Rs> has UNPREDICTABLE results.

Operand restriction <RdHi> and <RdLo> must be distinct registers, or the results are UNPREDICTABLE.

Specifying the same register for either <RdHi> and <Rm>, or <RdLo> and <Rm>, was previously described as producing UNPREDICTABLE results. There is no restriction in ARMv6, and it is believed all relevant ARMv4 and ARMv5 implementations do not require this restriction either, because high performance multipliers read all their operands prior to writing back any results.

Early termination If the multiplier implementation supports early termination, it must be implemented on the value of the <Rs> operand. The type of early termination used (signed or unsigned) is IMPLEMENTATION DEFINED.

C and V flags SMLALS is defined to leave the C and V flags unchanged in ARM architecture version 5 and above. In earlier versions of the architecture, the values of the C and V flags were UNPREDICTABLE after an SMLALS instruction.

A4.1.80 SMLAL<x><y>

31	28	27	26	25	24	23	22	21	20	19	16	15	12	11	8	7	6	5	4	3	0
cond		0	0	0	1	0	1	0	0	RdHi		RdLo		Rs		1	y	x	0	Rm	

SMLAL<x><y> (Signed Multiply-Accumulate Long BB, BT, TB, and TT) 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 sign-extended and added to the 64-bit accumulate value held in <RdHi> and <RdLo>, and the result is written back to <RdHi> and <RdLo>.

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}.

Syntax

SMLAL<x><y>{<cond>} <RdLo>, <RdHi>, <Rm>, <Rs>

where:

<x>	Specifies which half of the source register <Rm> is used as the first multiply operand. If <x> is B, then x == 0 in the instruction encoding and the bottom half (bits[15:0]) of <Rm> is used. If <x> is T, then x == 1 in the instruction encoding and the top half (bits[31:16]) of <Rm> is used.
<y>	Specifies which half of the source register <Rs> is used as the second multiply operand. If <y> is B, then y == 0 in the instruction encoding and the bottom half (bits[15:0]) of <Rs> is used. If <y> is T, then y == 1 in the instruction encoding and the top half (bits[31:16]) of <Rs> is used.
<cond>	Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-7. If <cond> is omitted, the AL (always) condition is used.
<RdLo>	Supplies the lower 32 bits of the 64-bit accumulate value to be added to the product, and is the destination register for the lower 32 bits of the 64-bit result.
<RdHi>	Supplies the upper 32 bits of the 64-bit accumulate value to be added to the product, and is the destination register for the upper 32 bits of the 64-bit result.
<Rm>	Specifies the source register whose bottom or top half (selected by <x>) is the first multiply operand.
<Rs>	Specifies the source register whose bottom or top half (selected by <y>) is the second multiply operand.

Architecture version

Version 6 and above, and version 5E.

Exceptions

None.

Operation

```
if ConditionPassed(cond) then

    if (x == 0) then
        operand1 = SignExtend(Rm[15:0])
    else /* x == 1 */
        operand1 = SignExtend(Rm[31:16])

    if (y == 0) then
        operand2 = SignExtend(Rs[15:0])
    else /* y == 1 */
        operand2 = SignExtend(Rs[31:16])

    RdLo = RdLo + (operand1 * operand2)
    RdHi = RdHi + (if (operand1*operand2) < 0 then 0xFFFFFFFF else 0)
                + CarryFrom(RdLo + (operand1 * operand2))
```

Usage

These instructions allow a long sequence of multiply-accumulates of signed 16-bit integers or Q15 numbers to be performed, with sufficient *guard bits* to ensure that the result cannot overflow the 64-bit destination in practice. It would take more than 2^{33} consecutive multiply-accumulates to cause such overflow.

If the overall calculation does not overflow a signed 32-bit number, then <RdLo> holds the result of the calculation.

A simple test to determine whether such a calculation has overflowed <RdLo> is to execute the instruction:

```
    CMP      <RdHi>, <RdLo>, ASR #31
```

at the end of the calculation. If the Z flag is set, <RdLo> holds an accurate final result. If the Z flag is clear, the final result has overflowed a signed 32-bit destination.

Notes

Use of R15   Specifying R15 for register <RdLo>, <RdHi>, <Rm>, or <Rs> has
                          UNPREDICTABLE results.

Operand restriction If <RdLo> and <RdHi> are the same register, the results are UNPREDICTABLE.

Condition flags   The SMLAL<x><y> instructions do not affect the N, Z, C, V, or Q flags.

A4.1.81 SMLALD

31	28	27 26 25 24 23 22 21 20	19	16	15	12	11	8	7	6	5	4	3	0
cond		0 1 1 1 0 1 0 0	RdHi		RdLo		Rs		0	0	X	1	Rm	

SMLALD (Signed Multiply Accumulate Long Dual) performs two signed 16 x 16-bit multiplications. It adds the products to a 64-bit accumulate operand.

Optionally, you can exchange the halfwords of the second operand before performing the arithmetic. This produces top ∞ bottom and bottom ∞ top multiplication.

Syntax

SMLALD{X}{<cond>}   <RdLo>, <RdHi>, <Rm>, <Rs>

where:

X   Sets the X bit of the instruction to 1, and the multiplications are bottom ∞ top and top ∞ bottom.

               If the X is omitted, sets the X bit to 0, and the multiplications are bottom ∞ bottom and top ∞ top.

<cond>   Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-7. If <cond> is omitted, the AL (always) condition is used.

<RdLo>   Supplies the lower 32 bits of the 64-bit accumulate value to be added to the product, and is the destination register for the lower 32 bits of the 64-bit result.

<RdHi>   Supplies the upper 32 bits of the 64-bit accumulate value to be added to the product, and is the destination register for the upper 32 bits of the 64-bit result.

<Rm>   Specifies the register that contains the first multiply operand.

<Rs>   Specifies the register that contains the second multiply operand.

Architecture version

Version 6 and above.

Exceptions

None.

Operation

```
if ConditionPassed(cond) then
    if X == 1 then
        operand2 = Rs Rotate_Right 16
    else
        operand2 = Rs
    accvalue[31:0]  = RdLo
    accvalue[63:32] = RdHi
    product1 = Rm[15:0]  * operand2[15:0]     /* Signed multiplication */
    product2 = Rm[31:16] * operand2[31:16]    /* Signed multiplication */
    result = accvalue + product1 + product2   /* Signed addition */
    RdLo = result[31:0]
    RdHi = result[63:32]
```

Usage

Use SMLALD in similar ways to SMLAD, but when you require a 64-bit accumulator instead of a 32 bit accumulator. On most implementations, this runs more slowly. See the usage section for *SMLAD* on page A4-152 for further details.

Notes

Use of R15   Specifying R15 for register <RdLo>, <RdHi>, <Rm>, or <Rs> has UNPREDICTABLE results.

Early termination   If the multiplier implementation supports early termination, it must be implemented on the value of the <Rs> operand. The type of early termination used (signed or unsigned) is IMPLEMENTATION DEFINED.

Flags   SMLALD leaves all the flags unchanged.

Confidential - Draft

A4.1.82 SMLAW<y>

31	28	27	26	25	24	23	22	21	20	19	16	15	12	11	8	7	6	5	4	3	0
cond		0	0	0	1	0	0	1	0	Rd		Rn		Rs		1	y	0	0	Rm	

SMLAW<y> (Signed Multiply-Accumulate Word B and T) performs a signed multiply-accumulate operation. The multiply acts on a signed 32-bit quantity and a signed 16-bit quantity, with the latter being 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 CPSR. No overflow can occur during the multiplication, because of the use of the top 32 bits of the 48-bit product.

Syntax

SMLAW<y>{<cond>} <Rd>, <Rm>, <Rs>, <Rn>

where:

<y>	Specifies which half of the source register <Rs> is used as the second multiply operand. If <y> is B, then y == 0 in the instruction encoding and the bottom half (bits[15:0]) of <Rs> is used. If <y> is T, then y == 1 in the instruction encoding and the top half (bits[31:16]) of <Rs> is used.
<cond>	Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-7. If <cond> is omitted, the AL (always) condition is used.
<Rd>	Specifies the destination register.
<Rm>	Specifies the source register which contains the 32-bit first multiply operand.
<Rs>	Specifies the source register whose bottom or top half (selected by <y>) is the second multiply operand.
<Rn>	Specifies the register which contains the accumulate value.

Architecture version

Version 6 and above, and version 5E.

Exceptions

None.

Operation

```
if ConditionPassed(cond) then

    if (y == 0) then
        operand2 = SignExtend(Rs[15:0])
    else /* y == 1 */
        operand2 = SignExtend(Rs[31:16])

    Rd = (Rm * operand2)[47:16] + Rn     /* Signed multiplication */
    if OverflowFrom((Rm * operand2)[47:16] + Rn) then
        Q Flag = 1
```

Usage

In addition to their straightforward uses for integer multiply-accumulates, these instructions sometimes provide a faster alternative to $Q31 \times Q15 + Q31 \rightarrow Q31$ multiply-accumulates synthesized from `SMULW<y>` and `QDADD` instructions. The circumstances under which this is possible and the benefits it provides are very similar to those for the `SMLA<x><y>` instructions. See *Usage* on page A4-151 for more details.

Notes

Use of R15   Specifying R15 for register `<Rd>`, `<Rm>`, `<Rs>`, or `<Rn>` has UNPREDICTABLE results.

Condition flags   The `SMLAW<y>` instructions do not affect the N, Z, C, or V flags.

A4.1.83 SMLSD

31 28	27 26 25 24 23 22 21 20	19 16	15 12	11 8	7 6 5 4	3 0
cond	0 1 1 1 0 0 0 0	Rd	Rn	Rs	0 1 X 1	Rm
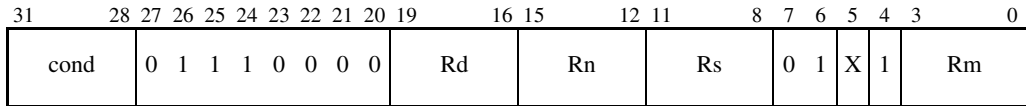
SMLSD (Signed Multiply Subtract accumulate Dual) performs two signed 16 x 16-bit multiplications. It adds the difference of the products to a 32-bit accumulate operand.

Optionally, you can exchange the halfwords of the second operand before performing the arithmetic. This produces top ∞ bottom and bottom ∞ top multiplication.

This instruction sets the Q flag if the accumulate operation overflows. Overflow cannot occur during the multiplications or subtraction.

Syntax

SMLSD{X}{<cond>}   <Rd>, <Rm>, <Rs>, <Rn>

where:

X	Sets the X bit of the instruction to 1, and the multiplications are bottom ∞ top and top ∞ bottom.
	If the X is omitted, sets the X bit to 0, and the multiplications are bottom ∞ bottom and top ∞ top.
<cond>	Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-7. If <cond> is omitted, the AL (always) condition is used.
<Rd>	Specifies the destination register.
<Rm>	Specifies the register that contains the first multiply operand.
<Rs>	Specifies the register that contains the second multiply operand.
<Rn>	Specifies the register that contains the accumulate operand.

Architecture version

Version 6 and above.

Exceptions

None.

Operation

```
if ConditionPassed(cond) then
   if X == 1 then
       operand2 = Rs Rotate_Right 16
   else
       operand2 = Rs
   product1 = Rm[15:0]  * operand2[15:0]     /* Signed multiplication */
   product2 = Rm[31:16] * operand2[31:16]    /* Signed multiplication */
   diffofproducts = product1 - product2      /* Signed subtraction */
   Rd = Rn + diffofproducts
   if OverflowFrom(Rn + diffofproducts) then
       Q flag = 1
```

Usage

You can use SMLSD for calculating the real part in filters with 32-bit accumulators, acting on complex numbers with 16-bit real and 16-bit imaginary parts.

See also the usage section for *SMLAD* on page A4-152.

Notes

Use of R15   Specifying R15 for register <Rd>, <Rm>, or <Rs> has UNPREDICTABLE results.

> ─── **Note** ───
> Your assembler must fault the use of R15 for register <Rn>.

Encoding   If the <Rn> field of the instruction contains 0b1111, the instruction is an SMUSD instruction instead, see *SMUSD* on page A4-180.

Early termination   If the multiplier implementation supports early termination, it must be implemented on the value of the <Rs> operand. The type of early termination used (signed or unsigned) is IMPLEMENTATION DEFINED.

C and V flags   SMLSD leaves the C and V flags unchanged.

Copyright © 1996–2003 ARM Limited. All rights reserved.
Confidential - Draft

A4.1.84 SMLSLD

31	28	27 26 25 24 23 22 21 20	19	16	15	12	11	8	7	6	5	4	3	0
cond		0 1 1 1 0 1 0 0	RdHi		RdLo		Rs		0	1	X	1	Rm	

SMLSLD (Signed Multiply Subtract accumulate Long Dual) performs two signed 16 x 16-bit multiplications. It adds the difference of the products to a 64-bit accumulate operand.

Optionally, you can exchange the halfwords of the second operand before performing the arithmetic. This produces top ∞ bottom and bottom ∞ top multiplication.

Syntax

SMLSLD{X}{<cond>} <RdLo>, <RdHi>, <Rm>, <Rs>

where:

X	Sets the X bit of the instruction to 1, and the multiplications are bottom ∞ top and top ∞ bottom.
	If the X is omitted, sets the X bit to 0, and the multiplications are bottom ∞ bottom and top ∞ top.
<cond>	Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-7. If <cond> is omitted, the AL (always) condition is used.
<RdLo>	Supplies the lower 32 bits of the 64-bit accumulate value to be added to the product, and is the destination register for the lower 32 bits of the 64-bit result.
<RdHi>	Supplies the upper 32 bits of the 64-bit accumulate value to be added to the product, and is the destination register for the upper 32 bits of the 64-bit result.
<Rm>	Specifies the register that contains the first multiply operand.
<Rs>	Specifies the register that contains the second multiply operand.

Architecture version

Version 6 and above.

Exceptions

None.

Operation

```
if ConditionPassed(cond) then
   if X == 1 then
       operand2 = Rs Rotate_Right 16
   else
       operand2 = Rs
   accvalue[31:0]  = RdLo
   accvalue[63:32] = RdHi
   product1 = Rm[15:0]  * operand2[15:0]    /* Signed multiplication */
   product2 = Rm[31:16] * operand2[31:16]   /* Signed multiplication */
   result = accvalue + product1 - product2  /* Signed subtraction */
   RdLo = result[31:0]
   RdHi = result[63:32]
```

Usage

The instruction has similar uses to those of the SMLSD instruction (see the Usage section for *SMLSD* on page A4-162), but when 64-bit accumulators are required rather than 32-bit accumulators. On most implementations, the resulting filter will not run as fast as a version using SMLSD, but it has many more guard bits against overflow.

See also the usage section for *SMLAD* on page A4-152.

Notes

Use of R15   Specifying R15 for register <Rd>, <Rm>, or <Rs> has UNPREDICTABLE results.

> ——— **Note** ———
>
> Your assembler must fault the use of R15 for register <Rn>.

Encoding   If the <Rn> field of the instruction contains 0b1111, the instruction is an SMUSD instruction instead, see *SMUSD* on page A4-180.

Early termination   If the multiplier implementation supports early termination, it must be implemented on the value of the <Rs> operand. The type of early termination used (signed or unsigned) is IMPLEMENTATION DEFINED.

C and V flags   SMLSD leaves the C and V flags unchanged.

A4.1.85 SMMLA
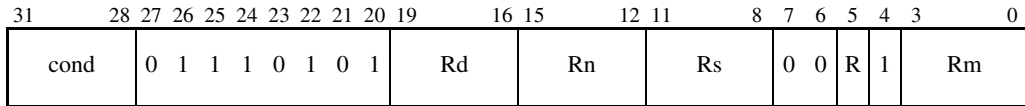
31	28	27 26 25 24 23 22 21 20	19	16	15	12	11	8	7	6	5	4	3	0
cond		0 1 1 1 0 1 0 1	Rd		Rn		Rs		0	0	R	1	Rm	

SMMLA (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, you 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.

Syntax

SMMLA{R}{<cond>}   <Rd>, <Rm>, <Rs>, <Rn>

where:

R	Sets the R bit of the instruction to 1. The multiplication is rounded.
	If the R is omitted, sets the R bit to 0. The multiplication is truncated.
<cond>	Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-7. If <cond> is omitted, the AL (always) condition is used.
<Rd>	Specifies the destination register.
<Rm>	Specifies the register that contains the first multiply operand.
<Rs>	Specifies the register that contains the second multiply operand.
<Rn>	Specifies the register that contains the accumulate operand.

Architecture version

Version 6 and above.

Exceptions

None.

ARM DDI 0100F-02

Operation

```
if ConditionPassed(cond) then
    value = Rm * Rs                      /* Signed multiplication */
    if R == 1 then
        Rd = ((Rn<<32) + value + 0x80000000)[63:32]
    else
        Rd = ((Rn<<32) + value)[63:32]
```

Usage

Provides fast multiplication for 32-bit fractional arithmetic. For example, the multiplies take two Q31 inputs and give a Q30 result (where Q*n* is a fixed point number with *n* bits of fraction).

Notes

Use of R15   Specifying R15 for register <Rd>, <Rm>, or <Rs> has UNPREDICTABLE results.

————— **Note** —————

Your assembler must fault the use of R15 for register <Rn>.

Encoding   If the <Rn> field of the instruction contains 0b1111, the instruction is an SMMUL instruction instead, see *SMMUL* on page A4-170.

Early termination   If the multiplier implementation supports early termination, it must be implemented on the value of the <Rs> operand. The type of early termination used (signed or unsigned) is IMPLEMENTATION DEFINED.

Flags   SMMLA leaves all the flags unchanged.

A4.1.86 SMMLS

31	28 27 26 25 24 23 22 21 20	19 16	15 12	11 8	7 6 5 4	3 0
cond	0 1 1 1 0 1 0 1	Rd	Rn	Rs	1 1 R 1	Rm
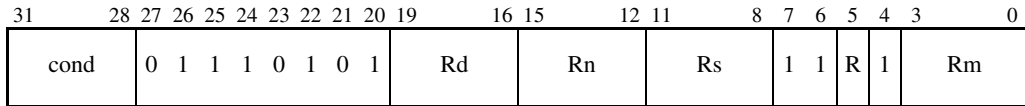
SMMLS (Signed Most significant word Multiply Subtract) multiplies two signed 32-bit values, extracts the most significant 32 bits of the result, and subtracts it from an accumulate value.

Optionally, you can specify that the result is rounded instead of being truncated. In this case, the constant 0x80000000 is added to the accumulated value before the high word is extracted.

Syntax

SMMLS{R}{<cond>} <Rd>, <Rm>, <Rs>, <Rn>

where:

R	Sets the R bit of the instruction to 1. The multiplication is rounded.
	If the R is omitted, sets the R bit to 0. The multiplication is truncated.
<cond>	Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-7. If <cond> is omitted, the AL (always) condition is used.
<Rd>	Specifies the destination register.
<Rm>	Specifies the register that contains the first multiply operand.
<Rs>	Specifies the register that contains the second multiply operand.
<Rn>	Specifies the register that contains the accumulate operand.

Architecture version

Version 6 and above.

Exceptions

None.

Operation

```
if ConditionPassed(cond) then
    value = Rm * Rs                    /* Signed multiplication */
    if R == 1 then
        Rd = ((Rn<<32) - value + 0x80000000)[63:32]
    else
        Rd = ((Rn<<32) – value)[63:32]
```

Usage

Provides fast multiplication for 32-bit fractional arithmetic. For example, the multiplies take two Q31 inputs and give a Q30 result (where Q*n* is a fixed point number with *n* bits of fraction).

Notes

Use of R15	Specifying R15 for register <Rd>, <Rm>, <Rs>, or <Rn> has UNPREDICTABLE results.
Early termination	If the multiplier implementation supports early termination, it must be implemented on the value of the <Rs> operand. The type of early termination used (signed or unsigned) is IMPLEMENTATION DEFINED.
Flags	SMMLS leaves all the flags unchanged.

A4.1.87 SMMUL

31	28	27	26	25	24	23	22	21	20	19	16	15	14	13	12	11	8	7	6	5	4	3	0
cond		0	1	1	1	0	1	0	1	Rd		1	1	1	1	Rs		0	0	R	1	Rm	
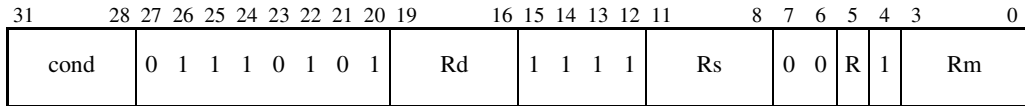
`SMMUL` (Signed Most significant word Multiply) multiplies two signed 32-bit values, and extracts the most significant 32 bits of the result.

Optionally, you 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.

Syntax

```
SMMUL{R}{<cond>}  <Rd>, <Rm>, <Rs>
```

where:

R	Sets the R bit of the instruction to 1. The multiplication is rounded.
	If the R is omitted, sets the R bit to 0. The multiplication is truncated.
`<cond>`	Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-7. If `<cond>` is omitted, the AL (always) condition is used.
`<Rd>`	Specifies the destination register.
`<Rm>`	Specifies the register that contains the first multiply operand.
`<Rs>`	Specifies the register that contains the second multiply operand.

Architecture version

Version 6 and above.

Exceptions

None.

Operation

```
if ConditionPassed(cond) then
    if R == 1 then
        value = Rm * Rs + 0x80000000  /* Signed multiplication */
    else
        value = Rm * Rs               /* Signed multiplication */
    Rd = value[63:32]
```

Usage

You can use SMMUL in combination with QADD or QDADD to perform Q31 multiplies and multiply-accumulates. It has two advantages over a combination of SMULL with QADD or QDADD:

*   you can round the product
*   no scratch register is required for the least significant half of the product.

You can also use SMMUL in optimized Fast Fourier Transforms and similar algorithms.

Notes

Use of R15 Specifying R15 for register <Rd>, <Rm>, or <Rs> has UNPREDICTABLE results.

Early termination If the multiplier implementation supports early termination, it must be implemented on the value of the <Rs> operand. The type of early termination used (signed or unsigned) is IMPLEMENTATION DEFINED.

Flags SMMUL leaves all the flags unchanged.

A4.1.88 SMUAD

31	28	27 26 25 24 23 22 21 20	19	16	15 14 13 12	11	8	7 6	5	4	3	0
cond		0 1 1 1 0 0 0 0	Rd		1 1 1 1	Rs		0 0	X	1	Rm	

SMUAD (Signed Dual Multiply Add) performs two signed 16 x 16-bit multiplications. It adds the products together, giving a 32-bit result.

Optionally, you can exchange the halfwords of the second operand before performing the arithmetic. This produces top ∞ bottom and bottom ∞ top multiplication.

This instruction sets the Q flag if the addition overflows. The multiplications cannot overflow.

Syntax

SMUAD{X}{<cond>} <Rd>, <Rm>, <Rs>

where:

X   Sets the X bit of the instruction to 1, and the multiplications are bottom ∞ top and top ∞ bottom.

              If the X is omitted, sets the X bit to 0, and the multiplications are bottom ∞ bottom and top ∞ top.

<cond>   Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-7. If <cond> is omitted, the AL (always) condition is used.

<Rd>   Specifies the destination register.

<Rm>   Specifies the register that contains the first operand.

<Rs>   Specifies the register that contains the second operand.

Architecture version

Version 6 and above.

Exceptions

None.

Operation

```
if ConditionPassed(cond) then
    if X == 1 then
        operand2 = Rs Rotate_Right 16
    else
        operand2 = Rs
    product1 = Rm[15:0]  * operand2[15:0]     /* Signed multiplication */
    product2 = Rm[31:16] * operand2[31:16]    /* Signed multiplication */
    Rd = product1 + product2
    if OverflowFrom(product1 + product2) then
        Q flag = 1
```

Usage

Use SMUAD for the first pair of multiplications in a sequence that uses the SMLAD instruction for the following multiplications, see *SMLAD* on page A4-152.

You can use the X option for calculating the imaginary part of a product of complex numbers with 16-bit real and 16-bit imaginary parts.

Notes

Use of R15   Specifying R15 for register `<Rd>`, `<Rm>`, or `<Rs>` has UNPREDICTABLE results.

Early termination   If the multiplier implementation supports early termination, it must be implemented on the value of the `<Rs>` operand. The type of early termination used (signed or unsigned) is IMPLEMENTATION DEFINED.

C and V flags   SMUAD leaves the C and V flags unchanged.

A4.1.89 SMUL<x><y>

31	28 27 26 25 24 23 22 21 20	19	16	15	12	11	8	7	6	5	4	3	0
cond	0 0 0 1 0 1 1 0	Rd		SBZ		Rs		1	y	x	0	Rm	

SMUL<x><y> (Signed Multiply BB, BT, TB, or TT) performs a signed multiply 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. No overflow is possible during this instruction.

Syntax

SMUL<x><y>{<cond>} <Rd>, <Rm>, <Rs>

where:

<x>	Specifies which half of the source register <Rm> is used as the first multiply operand. If <x> is B, then x == 0 in the instruction encoding and the bottom half (bits[15:0]) of <Rm> is used. If <x> is T, then x == 1 in the instruction encoding and the top half (bits[31:16]) of <Rm> is used.
<y>	Specifies which half of the source register <Rs> is used as the second multiply operand. If <y> is B, then y == 0 in the instruction encoding and the bottom half (bits[15:0]) of <Rs> is used. If <y> is T, then y == 1 in the instruction encoding and the top half (bits[31:16]) of <Rs> is used.
<cond>	Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-7. If <cond> is omitted, the AL (always) condition is used.
<Rd>	Specifies the destination register.
<Rm>	Specifies the source register whose bottom or top half (selected by <x>) is the first multiply operand.
<Rs>	Specifies the source register whose bottom or top half (selected by <y>) is the second multiply operand.

Architecture version

Version 6 and above, and version 5E.

Exceptions

None.

Operation

```
if ConditionPassed(cond) then

    if (x == 0) then
        operand1 = SignExtend(Rm[15:0])
    else /* x == 1 */
        operand1 = SignExtend(Rm[31:16])

    if (y == 0) then
        operand2 = SignExtend(Rs[15:0])
    else /* y == 1 */
        operand2 = SignExtend(Rs[31:16])

    Rd = operand1 * operand2
```

Usage

In addition to its straightforward uses for integer multiplies, this instruction can be used in combination with QADD, QDADD, and QDSUB to perform multiplies, multiply-accumulates, and multiply-subtracts on Q15 numbers. See the *Usage* sections on page A4-92, page A4-100, and page A4-102 for examples.

Notes

Use of R15   Specifying R15 for register `<Rd>`, `<Rm>`, or `<Rs>` has UNPREDICTABLE results.

Condition flags   SMUL`<x><y>` does not affect the N, Z, C, V, or Q flags.

A4.1.90 SMULL

31	28	27 26 25 24 23 22 21	20	19 16	15 12	11 8	7 6 5 4	3 0
cond		0 0 0 0 1 1 0	S	RdHi	RdLo	Rs	1 0 0 1	Rm
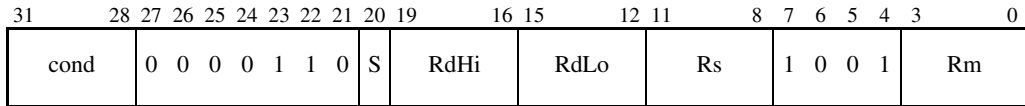
SMULL (Signed Multiply Long) multiplies two 32-bit signed values to produce a 64-bit result.

SMULL can optionally update the condition code flags, based on the 64-bit result.

Syntax

SMULL{<cond>}{S} <RdLo>, <RdHi>, <Rm>, <Rs>

where:

<cond>   Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-7. If <cond> is omitted, the AL (always) condition is used.

S   Causes the S bit (bit[20]) in the instruction to be set to 1 and specifies that the instruction updates the CPSR by setting the N and Z flags according to the result of the multiplication. If S is omitted, the S bit of the instruction is set to 0 and the entire CPSR is unaffected by the instruction.

<RdLo>   Stores the lower 32 bits of the result.

<RdHi>   Stores the upper 32 bits of the result.

<Rm>   Holds the signed value to be multiplied with the value of <Rs>.

<Rs>   Holds the signed value to be multiplied with the value of <Rm>.

Architecture version

Versions 3M and 4 and above, excluding xM versions.

Exceptions

None.

ARM DDI 0100F-02

Operation

```
if ConditionPassed(cond) then
    RdHi = (Rm * Rs)[63:32] /* Signed multiplication */
    RdLo = (Rm * Rs)[31:0]
    if S == 1 then
        N Flag = RdHi[31]
        Z Flag = if (RdHi == 0) and (RdLo == 0) then 1 else 0
        C Flag = unaffected    /* See "C and V flags" note */
        V Flag = unaffected    /* See "C and V flags" note */
```

Usage

SMULL multiplies signed variables to produce a 64-bit result in two general-purpose registers.

Notes

Use of R15   Specifying R15 for register <RdHi>, <RdLo>, <Rm>, or <Rs> has
                       UNPREDICTABLE results.

Operand restriction   <RdHi> and <RdLo> must be distinct registers, or the results are UNPREDICTABLE.

                       Specifying the same register for either <RdHi> and <Rm>, or <RdLo> and <Rm>,
                       was previously described as producing UNPREDICTABLE results. There is no
                       restriction in ARMv6, and it is believed all relevant ARMv4 and ARMv5
                       implementations do not require this restriction either, because high performance
                       multipliers read all their operands prior to writing back any results.

Early termination   If the multiplier implementation supports early termination, it must be implemented
                       on the value of the <Rs> operand. The type of early termination used (signed or
                       unsigned) is IMPLEMENTATION DEFINED.

C and V flags   SMULLS is defined to leave the C and V flags unchanged in ARM architecture
                       version 5 and above. In earlier versions of the architecture, the values of the C and
                       V flags were UNPREDICTABLE after an SMULLS instruction.

A4.1.91 SMULW<y>

31	28	27	26	25	24	23	22	21	20	19	16	15	12	11	8	7	6	5	4	3	0
cond		0	0	0	1	0	0	1	0	Rd		SBZ		Rs		1	y	1	0	Rm	

SMULW<y> (Signed Multiply Word B and T) performs a signed multiply operation. The multiply acts on a signed 32-bit quantity and a signed 16-bit quantity, with the latter being 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.

Syntax

```
SMULW<y>{<cond>}  <Rd>, <Rm>, <Rs>
```

where:

<y>	Specifies which half of the source register <Rs> is used as the second multiply operand. If <y> is B, then y == 0 in the instruction encoding and the bottom half (bits[15:0]) of <Rs> is used. If <y> is T, then y == 1 in the instruction encoding and the top half (bits[31:16]) of <Rs> is used.
<cond>	Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-7. If <cond> is omitted, the AL (always) condition is used.
<Rd>	Specifies the destination register.
<Rm>	Specifies the source register which contains the 32-bit first operand.
<Rs>	Specifies the source register whose bottom or top half (selected by <y>) is the second operand.

Architecture version

Version 6 and above, and version 5E.

Exceptions

None.

Operation

```
if ConditionPassed(cond) then

    if (y == 0) then
        operand2 = SignExtend(Rs[15:0])
    else /* y == 1 */
        operand2 = SignExtend(Rs[31:16])

    Rd = (Rm * operand2)[47:16] /* Signed multiplication */
```

Usage

In addition to its straightforward uses for integer multiplies, this instruction can be used in combination with
QADD, QDADD, and QDSUB to perform multiplies, multiply-accumulates and multiply-subtracts between
Q31 and Q15 numbers. See the *Usage* sections on page A4-92, page A4-100, and page A4-102 for
examples.

Notes

Use of R15   Specifying R15 for register <Rd>, <Rm>, or <Rs> has UNPREDICTABLE results.

Condition flags   SMULW<y> does not affect the N, Z, C, V, or Q flags.

A4.1.92 SMUSD

31	28	27 26 25 24 23 22 21 20	19	16	15 14 13 12	11	8	7 6 5 4	3	0
cond		0 1 1 1 0 0 0 0	Rd		1 1 1 1	Rs		0 1 X 1	Rm	

SMUSD (Signed Dual Multiply Subtract) performs two signed 16 x 16-bit multiplications. It subtracts one product from the other, giving a 32-bit result.

Optionally, you can exchange the halfwords of the second operand before performing the arithmetic. This produces top ∞ bottom and bottom ∞ top multiplication.

Overflow cannot occur.

Syntax

SMUSD{X}{<cond>} <Rd>, <Rm>, <Rs>

where:

X   Sets the X bit of the instruction to 1. The multiplications are bottom ∞ top and top ∞ bottom.

                 If the X is omitted, sets the X bit to 0. The multiplications are bottom ∞ bottom and top ∞ top.

<cond>   Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-7. If <cond> is omitted, the AL (always) condition is used.

<Rd>   Specifies the destination register.

<Rm>   Specifies the register that contains the first multiply operand.

<Rs>   Specifies the register that contains the second multiply operand.

Architecture version

Version 6 and above.

Exceptions

None.

Operation

```
if ConditionPassed(cond) then
    if X == 1 then
        operand2 = Rs Rotate_Right 16
    else
        operand2 = Rs
    product1 = Rm[15:0]  * operand2[15:0]      /* Signed multiplication */
    product2 = Rm[31:16] * operand2[31:16]     /* Signed multiplication */
    Rd = product1 - product2                   /* Signed subtraction */
```

Usage

You can use SMUSD for calculating the real part of a complex product of complex numbers with 16-bit real and 16-bit imaginary parts.

Notes

Use of R15   Specifying R15 for register <Rd>, <Rm>, or <Rs> has UNPREDICTABLE results.

Early termination   If the multiplier implementation supports early termination, it must be implemented on the value of the <Rs> operand. The type of early termination used (signed or unsigned) is IMPLEMENTATION DEFINED.

Flags   SMUSD leaves all the flags unchanged.

A4.1.93 SRS

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15 — 12	11	10	9	8	7 — 5	4 — 0
1	1	1	1	1	0	0	P	U	1	W	0	1	1	0	1	SBZ	0	1	0	1	SBZ	mode
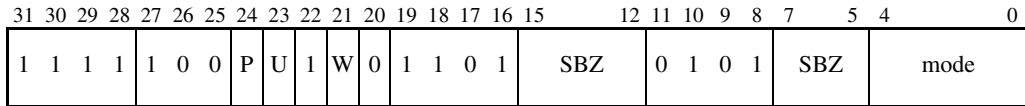
SRS (Store Return State) stores the R14 and SPSR of the current mode to the word at the specified address and the following word respectively. The address is determined from the banked version of R13 belonging to a specified mode.

Syntax

```
SRS<addressing_mode> #<mode>{!}
```

where:

`<addressing_mode>`

> Is similar to the `<addressing_mode>` in LDM and STM instructions, see *Addressing Mode 4 - Load and Store Multiple* on page A5-52, but with the following differences:
>
> - The base register, Rn, is the banked version of R13 for the mode specified by `<mode>`, rather than the current mode.
>
> - The number of registers to store is 2.
>
> - The register list is {R14, SPSR}, with both R14 and the SPSR being the versions belonging to the current mode.

`<mode>`   Specifies the number of the mode whose banked register is used as the base register for `<addressing_mode>`. The mode number is the 5-bit encoding of the chosen mode in a PSR, as described in *The control bits* on page A2-13.

`!`   If present, sets the W bit. This causes the instruction to write a modified value back to its base register, in a manner similar to that specified for *Addressing Mode 4 - Load and Store Multiple* on page A5-52. If `!` is omitted, the W bit is 0 and the instruction does not change the base register.

Architecture version

Version 6 and above.

Exceptions

Data Abort.

Operation

```
address = start_address
Memory[address,4] = R14
if CurrentModeHasSPSR() then
    Memory[address+4,4] = SPSR
else
    UNPREDICTABLE
assert end_address == address + 8
```

where `start_address` and `end_address` are determined as described in *Addressing Mode 4 - Load and Store Multiple* on page A5-52, with the following modifications:

• `Number_Of_Set_Bits_in(register_list)` evaluates to 2, rather than depending on bits[15:0] of the instruction.

• `Rn` is the banked version of `R13` belonging to the mode specified by the instruction, rather than being the version of `R13` of the current mode.

Notes

Data Abort   For details of the effects of this instruction if a Data Abort occurs, see *Data Abort (data access memory abort)* on page A2-20.

Non word-aligned addresses

SRS ignores the least significant two bits of `address`.

Alignment   If an implementation includes a System Control coprocessor (see Chapter B3 *The System Control Coprocessor*), and alignment checking is enabled, an address with bits[1:0] != 0b00 causes an alignment exception.

Time order   The time order of the accesses to individual words of memory generated by SRS is not architecturally defined. Do not use this instruction on memory-mapped I/O locations where access order matters.

User and System modes

SRS is UNPREDICTABLE in User and System modes, because they do not have SPSRs.

─── **Note** ───

In User mode, SRS must not give access to any banked registers belonging to other modes. This would constitute a security hole.

Condition   Unlike most other ARM instructions, SRS cannot be executed conditionally.

A4.1.94 SSAT
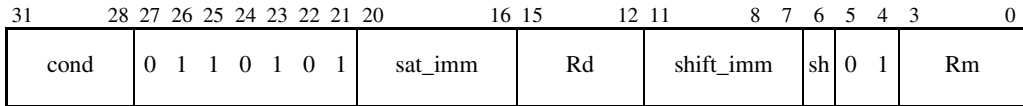
31	28	27 26 25 24 23 22 21	20 16	15 12	11 8	7 6	5 4	3 0
cond		0 1 1 0 1 0 1	sat_imm	Rd	shift_imm	sh	0 1	Rm

SSAT (Signed Saturate) saturates a signed value to a signed range. You can choose the bit position at which saturation occurs.

You can apply a shift to the value before the saturation occurs.

The Q flag is set if the operation saturates.

Syntax

```
SSAT{<cond>}  <Rd>, #<immed>, <Rm>{, <shift>}
```

where:

<cond>   Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-7. If <cond> is omitted, the AL (always) condition is used.

<Rd>   Specifies the destination register.

<immed>   Specifies the bit position for saturation, in the range 1 to 32. It is encoded in the sat_imm field of the instruction as <immed>-1.

<Rm>   Specifies the register that contains the value to be saturated.

<shift>   Specifies the optional shift. If present, it must be one of:

- LSL #*N*. *N* must be in the range 0 to 31.
  This is encoded as sh == 0 and shift_imm == *N*.

- ASR #*N*. *N* must be in the range 1 to 32. This is encoded as sh == 1 and either shift_imm == 0 for *N* == 32, or shift_imm == *N* otherwise.

If <shift> is omitted, LSL #0 is used.

Return

The value returned in Rd is:

$-2^{(n-1)}$ if X is $< -2^{(n-1)}$

X if $-2^{(n-1)} <= X <= 2^{(n-1)} - 1$

$2^{(n-1)} - 1$ if $X > 2^{(n-1)} - 1$

where n is <immed>, and X is the shifted value from Rm.

ARM DDI 0100F-02

Architecture version

Version 6 and above.

Exceptions

None.

Operation

```
if ConditionPassed(cond) then
    if shift == 1 then
        if shift_imm == 0 then
            operand = (Rm Artihmetic_Shift_Right 32)[31:0]
        else
            operand = (Rm Artihmetic_Shift_Right shift_imm)[31:0]
    else
        operand = (Rm Logical_Shift_Left shift_imm)[31:0]
    Rd = SignedSat(operand, sat_imm + 1)
    if SignedDoesSat(operand, sat_imm + 1) then
        Q Flag = 1
```

Usage

You can use SSAT in various DSP algorithms that require scaling and saturation of signed data.

Notes

Use of R15   Specifying R15 for register <Rd> or <Rm> has UNPREDICTABLE results.

A4.1.95 SSAT16
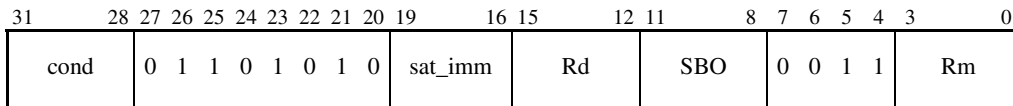
31 28	27 26 25 24 23 22 21 20	19 16	15 12	11 8	7 6 5 4	3 0
cond	0 1 1 0 1 0 1 0	sat_imm	Rd	SBO	0 0 1 1	Rm

SSAT16 saturates two 16-bit signed values to a signed range. You can choose the bit position at which saturation occurs.

Syntax

```
SSAT16{<cond>}  <Rd>, #<immed>, <Rm>
```

where:

<cond>   Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-7. If <cond> is omitted, the AL (always) condition is used.

<Rd>   Specifies the destination register.

<immed>   Specifies the bit position for saturation. This lies in the range 1 to 16. It is encoded in the sat_imm field of the instruction as <immed>-1.

<Rm>   Specifies the register that contains the values to be saturated.

Return

The value returned in each half of Rd is:

$-2^{(n-1)}$ if X is $< -2^{(n-1)}$

X if $-2^{(n-1)} <= X <= 2^{(n-1)} - 1$

$2^{(n-1)} - 1$ if X $> 2^{(n-1)} - 1$

where n is <immed>, and X is the value from the corresponding half of Rm.

Architecture version

Version 6 and above.

Exceptions

None.

Operation

```
if ConditionPassed(cond) then
    Rd[15:0]  = SignedSat(Rm[15:0],  sat_imm + 1)
    Rd[31:16] = SignedSat(Rm[31:16], sat_imm + 1)
    if SignedDoesSat(Rm[15:0],  sat_imm + 1)
                              OR SignedDoesSat(Rm[31:16], sat_imm + 1) then
        Q Flag = 1
```

Usage

You can use SSAT16 in various DSP algorithms that require saturation of signed data.

Notes

Use of R15   Specifying R15 for register <Rd> or <Rm> has UNPREDICTABLE results.

A4.1.96 SSUB16

31	28	27 26 25 24 23 22 21 20	19	16	15	12	11	8	7 6 5 4	3	0
cond		0 1 1 0 0 0 0 1	Rn		Rd		SBO		0 1 1 1	Rm	

SSUB16 (Signed Subtract) performs two 16-bit signed integer subtractions. It sets the GE bits in the CPSR according to the results of the subtractions.

Syntax

SSUB16{<cond>} <Rd>, <Rn>, <Rm>

where:

<cond>   Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-7. If <cond> is omitted, the AL (always) condition is used.

<Rd>   Specifies the destination register.

<Rn>   Specifies the register that contains the first operand.

<Rm>   Specifies the register that contains the second operand.

Architecture version

Version 6 and above.

Exceptions

None.

Operation

```
if ConditionPassed(cond) then
    diff      = Rn[15:0] - Rm[15:0]    /* Signed subtraction */
    Rd[15:0]  = diff[15:0]
    GE[1:0]   = if diff >= 0 then 0b11 else 0
    diff      = Rn[31:16] - Rm[31:16]  /* Signed subtraction */
    Rd[31:16] = diff[15:0]
    GE[3:2]   = if diff >= 0 then 0b11 else 0
```

Usage

Use SSUB16 to speed up operations on arrays of halfword data. This is similar to the way you can use SADD16. See the usage subsection for *SADD16* on page A4-120 for details.

You can also use `SSUB16` for operations on complex numbers that are held as pairs of 16-bit integers or Q15 numbers. If you hold the real and imaginary parts of a complex number in the bottom and top half of a register respectively, then the instruction:

```
SSUB16  Rd, Ra, Rb
```

performs the complex arithmetic operation Rd = Ra - Rb.

`SSUB16` sets the GE flags according to the results of each subtraction. You can use these in a following `SEL` instruction. See *SEL* on page A4-134 for further information.

Notes

Use of R15   Specifying R15 for register `<Rd>`, `<Rm>`, or `<Rn>` has UNPREDICTABLE results.

A4.1.97 SSUB8

31	28	27 26 25 24 23 22 21 20	19	16	15	12	11	8	7 6 5 4	3	0
cond		0 1 1 0 0 0 0 1	Rn		Rd		SBO		1 1 1 1	Rm	

SSUB8 performs four 8-bit signed integer subtractions. It sets the GE bits in the CPSR according to the results of the subtractions.

Syntax

SSUB8{<cond>} <Rd>, <Rn>, <Rm>

where:

<cond>   Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-7. If <cond> is omitted, the AL (always) condition is used.

<Rd>   Specifies the destination register.

<Rn>   Specifies the register that contains the first operand.

<Rm>   Specifies the register that contains the second operand.

Architecture version

Version 6 and above.

Exceptions

None.

Operation

```
if ConditionPassed(cond) then
    diff      = Rn[7:0] - Rm[7:0]      /* Signed subtraction */
    Rd[7:0]   = diff[7:0]
    GE[0]     = if diff >= 0 then 1 else 0
    diff      = Rn[15:8] - Rm[15:8]    /* Signed subtraction */
    Rd[15:8]  = diff[7:0]
    GE[1]     = if diff >= 0 then 1 else 0
    diff      = Rn[23:16] - Rm[23:16]  /* Signed subtraction */
    Rd[23:16] = diff[7:0]
    GE[2]     = if diff >= 0 then 1 else 0
    diff      = Rn[31:24] - Rm[31:24]  /* Signed subtraction */
    Rd[31:24] = diff[7:0]
    GE[3]     = if diff >= 0 then 1 else 0
```

Usage

Use `SSUB8` to speed up operations on arrays of byte data. This is similar to the way you can use `SADD16` to speed up operations on halfword data. See the usage subsection for *SADD16* on page A4-120 for details.

Notes

Use of R15   Specifying R15 for register `<Rd>`, `<Rm>`, or `<Rn>` has UNPREDICTABLE results.

A4.1.98 SSUBADDX

31	28	27	26	25	24	23	22	21	20	19	16	15	12	11	8	7	6	5	4	3	0
cond		0	1	1	0	0	0	0	1	Rn		Rd		SBO		0	1	0	1	Rm	

`SSUBADDX` (Signed Subtract and Add with Exchange) performs one 16-bit signed integer addition and one 16-bit signed integer subtraction. It exchanges the two halfwords of the second operand before it performs the arithmetic.

`SSUBADDX` sets the GE bits in the CPSR according to the results.

Syntax

```
SSUBADDX{<cond>}  <Rd>, <Rn>, <Rm>
```

where:

<cond>   Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-7. If `<cond>` is omitted, the AL (always) condition is used.

<Rd>   Specifies the destination register.

<Rn>   Specifies the register that contains the first operand.

<Rm>   Specifies the register that contains the second operand.

Architecture version

Version 6 and above.

Exceptions

None.

Operation

```
if ConditionPassed(cond) then
    sum       = Rn[15:0]  + Rm[31:16]    /* Signed addition */
    Rd[15:0]  = sum[16:1]
    GE[1:0]   = if sum >= 0 then 0b11 else 0
    diff      = Rn[31:16] - Rm[15:0]     /* Signed subtraction */
    Rd[31:16] = diff[16:1]
    GE[3:2]   = if diff >= 0 then 0b11 else 0
```

Usage

You can use SSUBADDX for operations on complex numbers that are held as pairs of 16-bit integers or Q15 numbers. If you hold the real and imaginary parts of a complex number in the bottom and top half of a register respectively, then the instruction:

```
SSUBADDX Rd, Ra, Rb
```

performs the complex arithmetic operation Rd = Ra - i * Rb.

Notes

Use of R15   Specifying R15 for register `<Rd>`, `<Rm>`, or `<Rn>` has UNPREDICTABLE results.

A4.1.99 STC

31	28	27	26	25	24	23	22	21	20	19	16	15	12	11	8	7	0
cond		1	1	0	P	U	N	W	0	Rn		CRd		cp_num		8_bit_word_offset	
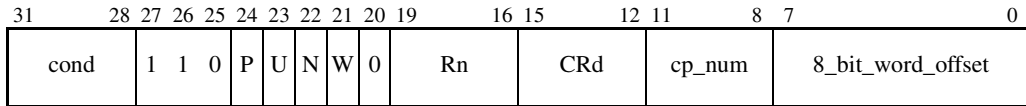
STC (Store Coprocessor) stores data from a coprocessor to a sequence of consecutive memory addresses. If no coprocessors indicate that they can execute the instruction, an Undefined Instruction exception is generated.

Syntax

```
STC{<cond>}{L}  <coproc>, <CRd>, <addressing_mode>
STC2{L}         <coproc>, <CRd>, <addressing_mode>
```

where:

<cond>   Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-7. If <cond> is omitted, the AL (always) condition is used.

STC2   Causes the condition field of the instruction to be set to 0b1111. This provides additional opcode space for coprocessor designers. The resulting instructions can only be executed unconditionally.

L   Sets the N bit (bit[22]) in the instruction to 1 and specifies a long store (for example, double-precision instead of single-precision data transfer). If L is omitted, the N bit is 0 and the instruction specifies a short store.

<coproc>   Specifies the name of the coprocessor, and causes the corresponding coprocessor number to be placed in the cp_num field of the instruction. The standard generic coprocessor names are p0, p1, ..., p15.

<CRd>   Specifies the coprocessor source register.

<addressing_mode>

                Is described in *Addressing Mode 5 - Load and Store Coprocessor* on page A5-60. It determines the P, U, Rn, W and 8_bit_word_offset bits of the instruction.

                The syntax of all forms of <addressing_mode> includes a *base register* <Rn>. Some forms also specify that the instruction modifies the base register value (this is known as *base register write-back*).

Architecture version

STC is in version 2 and above.

STC2 is in version 5 and above.

Exceptions

Undefined Instruction, Data Abort.

Operation

```
if ConditionPassed(cond) then
    address = start_address
    Memory[address,4] = value from Coprocessor[cp_num]
    while (NotFinished(coprocessor[cp_num]))
        address = address + 4
        Memory[address,4] = value from Coprocessor[cp_num]
    assert address == end_address
```

Usage

STC is useful for storing coprocessor data to memory. The L (long) option controls the N bit and could be used to distinguish between a single- and double-precision transfer for a floating-point store instruction.

Notes

Coprocessor fields   Only instruction bits[31:23], bits[21:16} and bits[11:0] are defined by the ARM architecture. The remaining fields (bit[22] and bits[15:12]) are recommendations, for compatibility with ARM Development Systems.

In the case of the Unindexed addressing mode (P==0, U==1, W==0), instruction bits[7:0] are also not ARM architecture-defined, and can be used to specify additional coprocessor options.

Data Abort   For details of the effects of the instruction if a Data Abort occurs, see *Effects of data-aborted instructions* on page A2-21.

Non word-aligned addresses

Store coprocessor register instructions ignore the least significant two bits of address.

Alignment   If an implementation includes a System Control coprocessor (see Chapter B3 *The System Control Coprocessor*), and alignment checking is enabled, an address with bits[1:0] != 0b00 causes an alignment exception.

Unimplemented coprocessor instructions

Hardware coprocessor support is optional, regardless of the architecture version. An implementation can choose to implement a subset of the coprocessor instructions, or no coprocessor instructions at all. Any coprocessor instructions that are not implemented instead cause an undefined instruction exception.

A4.1.100 STM (1)

31	28	27 26 25	24	23	22	21	20	19 16	15 0
cond		1 0 0	P	U	0	W	0	Rn	register_list
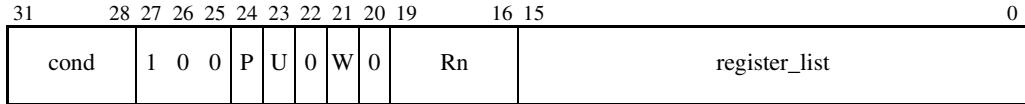
STM (1) (Store Multiple) stores a non-empty subset (or possibly all) of the general-purpose registers to sequential memory locations.

Syntax

STM{<cond>}<addressing_mode> <Rn>{!}, <registers>

where:

<cond>   Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-7. If <cond> is omitted, the AL (always) condition is used.

<addressing_mode>

                   Is described in *Addressing Mode 4 - Load and Store Multiple* on page A5-52. It determines the P, U, and W bits of the instruction.

<Rn>   Specifies the base register used by <addressing_mode>. If R15 is specified as <Rn>, the result is UNPREDICTABLE.

!   Sets the W bit, causing the instruction to write a modified value back to its base register Rn as specified in *Addressing Mode 4 - Load and Store Multiple* on page A5-52. If ! is omitted, the W bit is 0 and the instruction does not change its base register in this way.

<registers>   Is a list of registers, separated by commas and surrounded by { and }. It specifies the set of registers to be stored by the STM instruction.

                   The registers are stored in sequence, the lowest-numbered register to the lowest memory address (start_address), through to the highest-numbered register to the highest memory address (end_address).

                   For each of i=0 to 15, bit[i] in the register_list field of the instruction is 1 if Ri is in the list and 0 otherwise. If bits[15:0] are all zero, the result is UNPREDICTABLE.

                   If R15 is specified in <registers>, the value stored is IMPLEMENTATION DEFINED. For more details, see *Reading the program counter* on page A2-8.

Architecture version

All.

Exceptions

Data Abort.

Operation

```
if ConditionPassed(cond) then
    address = start_address
    for i = 0 to 15
        if register_list[i] == 1
            Memory[address,4] = Ri
            address = address + 4
    assert end_address == address - 4
```

Usage

STM is useful as a block store instruction (combined with LDM it allows efficient block copy) and for stack operations. A single STM used in the sequence of a procedure can push the return address and general-purpose register values on to the stack, updating the stack pointer in the process.

Notes

Operand restrictions

If <Rn> is specified as <registers> and base register write-back is specified:

• If <Rn> is the lowest-numbered register specified in <register_list>, the original value of <Rn> is stored.

• Otherwise, the stored value of <Rn> is UNPREDICTABLE.

Data Abort For details of the effects of the instruction if a Data Abort occurs, see *Effects of data-aborted instructions* on page A2-21.

Non word-aligned addresses

STM instructions ignore the least significant two bits of address.

Alignment If an implementation includes a System Control coprocessor (see Chapter B3 *The System Control Coprocessor*), and alignment checking is enabled, an address with bits[1:0] != 0b00 causes an alignment exception.

Time order The time order of the accesses to individual words of memory generated by this instruction is only defined in some circumstances. See *Data accesses to memory-mapped I/O* on page B2-27 for details.

A4.1.101 STM (2)

31 28	27 26 25	24	23	22	21	20	19 16	15 0
cond	1 0 0	P	U	1	0	0	Rn	register_list
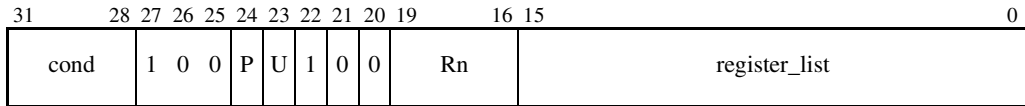
STM (2) stores a subset (or possibly all) of the User mode general-purpose registers to sequential memory locations.

Syntax

STM{<cond>}<addressing_mode> <Rn>, <registers>ˆ

where:

<cond>   Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-7. If <cond> is omitted, the AL (always) condition is used.

<addressing_mode>

                Is described in *Addressing Mode 4 - Load and Store Multiple* on page A5-52. It determines the P and U bits of the instruction. Only the forms of this addressing mode with W == 0 are available for this form of the STM instruction.

<Rn>   Specifies the base register used by <addressing_mode>. If R15 is specified as the base register <Rn>, the result is UNPREDICTABLE.

<registers>   Is a list of registers, separated by commas and surrounded by { and }. It specifies the set of registers to be stored by the STM instruction.

                The registers are stored in sequence, the lowest-numbered register to the lowest memory address (start_address), through to the highest-numbered register to the highest memory address (end_address).

                For each of i=0 to 15, bit[i] in the register_list field of the instruction is 1 if Ri is in the list and 0 otherwise. If bits[15:0] are all zero, the result is UNPREDICTABLE.

                If R15 is specified in <registers> the value stored is IMPLEMENTATION DEFINED. For more details, see *Reading the program counter* on page A2-8.

ˆ   For an STM instruction, indicates that User mode registers are to be stored.

Architecture version

All.

Exceptions

Data Abort.

Operation

```
if ConditionPassed(cond) then
    address = start_address
    for i = 0 to 15
        if register_list[i] == 1
            Memory[address,4] = Ri_usr
            address = address + 4
    assert end_address == address - 4
```

Usage

Use STM (2) to store the User mode registers when the processor is in a privileged mode (useful when performing process swaps, and in instruction emulators).

Notes

Write-back   Setting bit 21, the W bit, has UNPREDICTABLE results.

User and System mode

                    This instruction is UNPREDICTABLE in User or System mode.

Base register mode   For the purpose of address calculation, the base register is read from the current processor mode registers, not the User mode registers.

Data Abort   For details of the effects of the instruction if a Data Abort occurs, see *Effects of data-aborted instructions* on page A2-21.

Non word-aligned addresses

                    STM instructions ignore the least significant two bits of address.

Alignment   If an implementation includes a System Control coprocessor (see Chapter B3 *The System Control Coprocessor*), and alignment checking is enabled, an address with bits[1:0] != 0b00 causes an alignment exception.

Time order   The time order of the accesses to individual words of memory generated by this instruction is only defined in some circumstances. See *Data accesses to memory-mapped I/O* on page B2-27 for details.

Banked registers   In ARM architecture v5 and below, this form of LDM must not be followed by an instruction that accesses banked registers (a following NOP is a good way to ensure this).

A4.1.102 STR

31	28	27 26	25	24	23	22	21	20	19	16	15	12	11	0
cond		0 1	I	P	U	0	W	0	Rn		Rd		addr_mode	
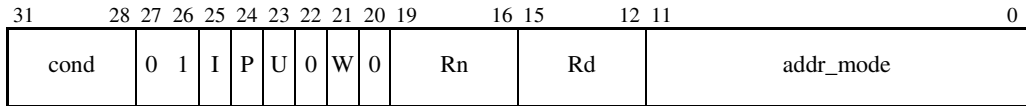
STR (Store Register) stores a word from a register to memory.

Syntax

```
STR{<cond>}  <Rd>, <addressing_mode>
```

where:

<cond>   Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-7. If <cond> is omitted, the AL (always) condition is used.

<Rd>   Specifies the source register for the operation. If R15 is specified for <Rd>, the value stored is IMPLEMENTATION DEFINED. For more details, see *Reading the program counter* on page A2-8.

<addressing_mode>

                Is described in *Addressing Mode 2 - Load and Store Word or Unsigned Byte* on page A5-19. It determines the I, P, U, W, Rn and addr_mode bits of the instruction.

                The syntax of all forms of <addressing_mode> includes a *base register* <Rn>. Some forms also specify that the instruction modifies the base register value (this is known as *base register write-back*).

Architecture version

All.

Exceptions

Data Abort.

Operation

```
if ConditionPassed(cond) then
    Memory[address,4] = Rd
```

Usage

Combined with a suitable addressing mode, STR stores 32-bit data from a general-purpose register into memory. Using the PC as the base register allows PC-relative addressing, which facilitates position-independent code.

Notes

Operand restrictions

If `<addressing_mode>` specifies base register write-back, and the same register is specified for `<Rd>` and `<Rn>`, the results are UNPREDICTABLE.

Data Abort For details of the effects of the instruction if a Data Abort occurs, see *Effects of data-aborted instructions* on page A2-21.

Non word-aligned addresses

`STR` instructions ignore the least significant two bits of `address`. So if these bits are not 0b00, the effects of `STR` are not precisely opposite to those of `LDR`.

Alignment If an implementation includes a System Control coprocessor (see Chapter B3 *The System Control Coprocessor*), and alignment checking is enabled, an address with bits[1:0] != 0b00 causes an alignment exception.

A4.1.103 STRB

31	28	27	26	25	24	23	22	21	20	19	16	15	12	11	0
cond		0	1	I	P	U	1	W	0	Rn		Rd		addr_mode	
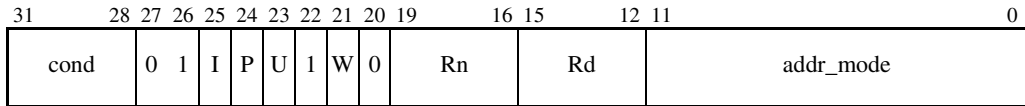
STRB (Store Register Byte) stores a byte from the least significant byte of a register to memory.

Syntax

```
STR{<cond>}B  <Rd>, <addressing_mode>
```

where:

<cond>   Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-7. If <cond> is omitted, the AL (always) condition is used.

<Rd>   Specifies the source register for the operation. If R15 is specified for <Rd>, the result is UNPREDICTABLE.

<addressing_mode>

        Is described in *Addressing Mode 2 - Load and Store Word or Unsigned Byte* on page A5-19. It determines the I, P, U, W, Rn and addr_mode bits of the instruction.

        The syntax of all forms of <addressing_mode> includes a *base register* <Rn>. Some forms also specify that the instruction modifies the base register value (this is known as *base register write-back*).

Architecture version

All.

Exceptions

Data Abort.

Operation

```
if ConditionPassed(cond) then
    Memory[address,1] = Rd[7:0]
```

Usage

Combined with a suitable addressing mode, STRB writes the least significant byte of a general-purpose register to memory. Using the PC as the base register allows PC-relative addressing, which facilitates position-independent code.

Notes

Operand restrictions

If `<addressing_mode>` specifies base register write-back, and the same register is specified for `<Rd>` and `<Rn>`, the results are UNPREDICTABLE.
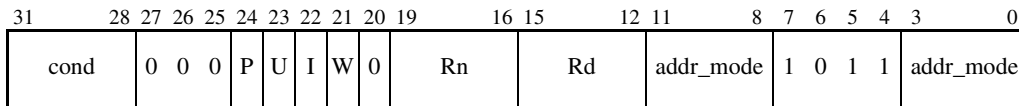
Data Abort   For details of the effects of the instruction if a Data Abort occurs, see *Effects of data-aborted instructions* on page A2-21.

A4.1.104 STRBT

31		28	27	26	25	24	23	22	21	20	19		16	15		12	11		0
cond			0	1	I	0	U	1	1	0	Rn			Rd			addr_mode		
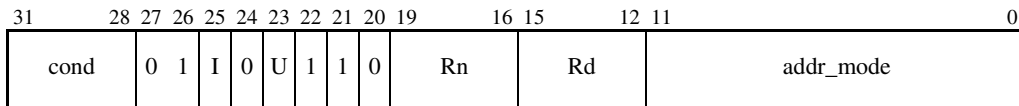
STRBT (Store Register Byte with Translation) stores a byte from the least significant byte of a register to memory. If the instruction is executed when the processor is in a privileged mode, the memory system is signaled to treat the access as if the processor were in User mode.

Syntax

```
STR{<cond>}BT  <Rd>, <post_indexed_addressing_mode>
```

where:

<cond>   Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-7. If <cond> is omitted, the AL (always) condition is used.

<Rd>   Specifies the source register for the operation. If R15 is specified for <Rd>, the result is UNPREDICTABLE.

<post_indexed_addressing_mode>

Is described in *Addressing Mode 2 - Load and Store Word or Unsigned Byte* on page A5-19. It determines the I, U, Rn and addr_mode bits of the instruction. Only post-indexed forms of Addressing Mode 2 are available for this instruction. These forms have P == 0 and W == 0, where P and W are bit[24] and bit[21] respectively. This instruction uses P == 0 and W == 1 instead, but the addressing mode is the same in all other respects.

The syntax of all forms of <post_indexed_addressing_mode> includes a *base register* <Rn>. All forms also specify that the instruction modifies the base register value (this is known as *base register write-back*).

Architecture version

All.

Exceptions

Data Abort.

Operation

```
if ConditionPassed(cond) then
    Memory[address,1] = Rd[7:0]
```

ARM DDI 0100F-02

Usage

STRBT can be used by a (privileged) exception handler that is emulating a memory access instruction which would normally execute in User mode. The access is restricted as if it had User mode privilege.

Notes

User mode   If this instruction is executed in User mode, an ordinary User mode access is performed.

Operand restrictions

If the same register is specified for <Rd> and <Rn>, the results are UNPREDICTABLE.

Data Abort   For details of the effects of the instruction if a Data Abort occurs, see *Effects of data-aborted instructions* on page A2-21.

Copyright © 1996–2003 ARM Limited. All rights reserved.
Confidential - Draft

A4.1.105 STRD

31	28	27	26	25	24	23	22	21	20	19	16	15	12	11	8	7	6	5	4	3	0
cond		0	0	0	P	U	I	W	0	Rn		Rd		addr_mode		1	1	1	1	addr_mode	

STRD (Store Registers Doubleword) stores a pair of ARM registers to two consecutive words of memory. The pair of registers is restricted to being an even-numbered register and the odd-numbered register that immediately follows it (for example, R10 and R11).

A greater variety of addressing modes is available than for a 2-register STM. However, the address of the first of the two words is required to be doubleword-aligned (that is, the address must be divisible by 8).

Syntax

```
STR{<cond>}D  <Rd>, <addressing_mode>
```

where:

<cond>   Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-7. If <cond> is omitted, the AL (always) condition is used.

<Rd>   Specifies the even-numbered register that is stored to the memory word addressed by <addressing_mode>. The immediately following odd-numbered register is stored to the next memory word. If <Rd> is R14, which would specify R15 as the second source register, the instruction is UNPREDICTABLE.

            If <Rd> specifies an odd-numbered register, the instruction is UNDEFINED.

<addressing_mode>

            Is described in *Addressing Mode 3 - Miscellaneous Loads and Stores* on page A5-38. It determines the P, U, I, W, Rn, and addr_mode bits of the instruction.

            The syntax of all forms of <addressing_mode> includes a *base register* <Rn>. Some forms also specify that the instruction modifies the base register value (this is known as *base register write-back*).

            The address generated by <addressing_mode> is the address of the lower of the two words stored by the STRD instruction. The address of the higher word is generated by adding 4 to this address.

Architecture version

Version 6 and above, and version 5E, but not ARMv5TExP.

Exceptions

Data Abort.

Operation

```
if ConditionPassed(cond) then
    if (Rd is even-numbered) then
        if (address[2:0] == 0b000) and (Rd is not R14) then
            Memory[address,4] = Rd
            Memory[address+4,4] = R(d+1)
        else
            UNPREDICTABLE
    else
        UNDEFINED
```

Notes

Operand restrictions

If `<addressing_mode>` performs base register write-back and the base register `<Rn>` is one of the two source registers of the instruction, the results are UNPREDICTABLE.

Data Abort For details of the effects of the instruction if a Data Abort occurs, see *Effects of data-aborted instructions* on page A2-21.

Non doubleword-aligned addresses

If the store address is not doubleword-aligned, the results are UNPREDICTABLE.

Alignment If an implementation includes a System Control coprocessor (see Chapter B3 *The System Control Coprocessor*), and alignment checking is enabled, then:

- an address with bits[1:0] != 0b00 causes an alignment exception

- an address with bits[2:0] == 0b000 does not cause an alignment exception

- it is IMPLEMENTATION DEFINED whether an address with bits[2:0] == 0b100 causes an alignment exception.

Time order The time order of the accesses to the two memory words is not architecturally defined. In particular, an implementation is allowed to perform the two 32-bit memory accesses in either order, or to combine them into a single 64-bit memory access.

A4.1.106 STREX

31	28	27	26	25	24	23	22	21	20	19	16	15	12	11	8	7	6	5	4	3	0
cond		0	0	0	1	1	0	0	0	Rn		Rd		SBO		1	0	0	1	Rm	

STREX (Store Register Exclusive) performs a conditional store to memory. The store only occurs if the executing processor has exclusive access to the memory addressed.

Syntax

```
STREX{<cond>} <Rd>, <Rm>, [<Rn>]
```

where:

<cond>   Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-7. If <cond> is omitted, the AL (always) condition is used.

<Rd>   Specifies the destination register for the returned status value. The value returned is:

    0   if the operation updates memory

    1   if the operation fails to update memory.

<Rm>   Specifies the register containing the word to be stored to memory.

<Rn>   Specifies the register containing the address.

Architecture version

Version 6 and above.

Exceptions

Data Abort.

Operation

```
if ConditionPassed(cond) then
    processor_id = ExecutingProcessor()
        if Shared(Rn) == 1 then
            physical_address = TLB(Rn)
            if IsExclusiveGlobal(physical_address,processor_id) then
                Memory[Rn,4] = Rm
                Rd = 0
                    ClearByAddress(physical_address)
            else
                Rd = 1
                ClearByProcessor(processor_id)
        endif
    else
        if IsExclusiveLocal(processor_id) then
            Memory[Rn,4] = Rm
            Rd = 0
        else
            Rd = 1
        endif
            ClearByProcessor(processor_id)
    endif
```

Usage

Use STREX in combination with LDREX to implement inter-process communication in multiprocessor and shared memory systems. See *LDREX* on page A4-51 for further information.

Notes

Use of R15   Specifying R15 for register <Rd>, <Rn>, or <Rm> has UNPREDICTABLE results.

Operand restrictions

<Rd> must be distinct from both <Rm> and <Rn>, otherwise the results are UNPREDICTABLE.

Data Abort   For details of the effects of the instruction if a Data Abort occurs, see *Effects of data-aborted instructions* on page A2-21. If a Data Abort occurs during execution of a STREX instruction:

- memory is not updated
- <Rd> is not updated.

Alignment   If <Rn>[1:0] != 0b00, then:

- If the implementation includes a System Control coprocessor (see Chapter B3 *The System Control Coprocessor*), and alignment checking is enabled, an alignment exception occurs.

- Otherwise, the result is UNPREDICTABLE.

A4.1.107 STRH

31	28	27	26	25	24	23	22	21	20	19	16	15	12	11	8	7	6	5	4	3	0
cond		0	0	0	P	U	I	W	0	Rn		Rd		addr_mode		1	0	1	1	addr_mode	

STRH (Store Register Halfword) stores a halfword from the least significant halfword of a register to memory. If the address is not halfword-aligned, the result is UNPREDICTABLE.

Syntax

```
STR{<cond>}H  <Rd>, <addressing_mode>
```

where:

<cond>   Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-7. If <cond> is omitted, the AL (always) condition is used.

<Rd>   Specifies the source register for the operation. If R15 is specified for <Rd>, the result is UNPREDICTABLE.

<addressing_mode>

Is described in *Addressing Mode 3 - Miscellaneous Loads and Stores* on page A5-38. It determines the P, U, I, W, Rn and addr_mode bits of the instruction.

The syntax of all forms of <addressing_mode> includes a *base register* <Rn>. Some forms also specify that the instruction modifies the base register value (this is known as *base register write-back*).

Architecture version

Version 4 and above.

Exceptions

Data Abort.

Operation

```
if ConditionPassed(cond) then
    if address[0] == 0
        data = Rd[15:0]
    else /* address[0] == 1 */
        data = UNPREDICTABLE
    Memory[address,2] = data
```

Usage

Combined with a suitable addressing mode, STRH allows 16-bit data from a general-purpose register to be stored to memory. Using the PC as the base register allows PC-relative addressing, to facilitate position-independent code.

Notes

Operand restrictions If `<addressing_mode>` specifies base register write-back, and the same register is specified for `<Rd>` and `<Rn>`, the results are UNPREDICTABLE.

Data Abort For details of the effects of the instruction if a Data Abort occurs, see *Effects of data-aborted instructions* on page A2-21.

Non halfword-aligned addresses

If the store address is not halfword-aligned, the stored value is UNPREDICTABLE.

Alignment If an implementation includes a System Control coprocessor (see Chapter B3 *The System Control Coprocessor*), and alignment checking is enabled, an address with bit[0] != 0 causes an alignment exception.

A4.1.108 STRT

31	28	27	26	25	24	23	22	21	20	19	16	15	12	11	0
cond		0	1	I	0	U	0	1	0	Rn		Rd		addr_mode	
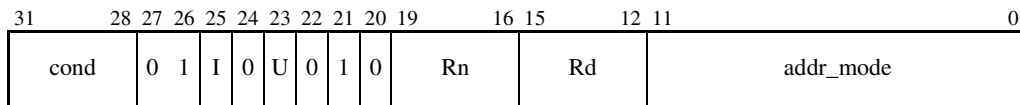
STRT (Store Register with Translation) stores a word from a register to memory. If the instruction is executed when the processor is in a privileged mode, the memory system is signaled to treat the access as if the processor was in User mode.

Syntax

STR{<cond>}T <Rd>, <post_indexed_addressing_mode>

where:

<cond>   Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-7. If <cond> is omitted, the AL (always) condition is used.

<Rd>   Specifies the source register for the operation. If R15 is specified for <Rd>, the value stored is IMPLEMENTATION DEFINED. For more details, see *Reading the program counter* on page A2-8.

<post_indexed_addressing_mode>

Is described in *Addressing Mode 2 - Load and Store Word or Unsigned Byte* on page A5-19. It determines the I, U, Rn and addr_mode bits of the instruction. Only post-indexed forms of Addressing Mode 2 are available for this instruction. These forms have P == 0 and W == 0, where P and W are bit[24] and bit[21] respectively. This instruction uses P == 0 and W == 1 instead, but the addressing mode is the same in all other respects.

The syntax of all forms of <post_indexed_addressing_mode> includes a *base register* <Rn>. All forms also specify that the instruction modifies the base register value (this is known as *base register write-back*).

Architecture version

All.

Exceptions

Data Abort.

Operation

```
if ConditionPassed(cond) then
    Memory[address,4] = Rd
```

Usage

STRT can be used by a (privileged) exception handler that is emulating a memory access instruction that would normally execute in User mode. The access is restricted as if it had User mode privilege.

Notes

User mode   If this instruction is executed in User mode, an ordinary User mode access is performed.

Operand restrictions

If the same register is specified for <Rd> and <Rn>, the results are UNPREDICTABLE.

Data Abort   For details of the effects of the instruction if a Data Abort occurs, see *Effects of data-aborted instructions* on page A2-21.

Alignment   If an implementation includes a System Control coprocessor (see Chapter B3 *The System Control Coprocessor*), and alignment checking is enabled, an address with bits[1:0] != 0b00 causes an alignment exception.

A4.1.109 SUB

31	28	27	26	25	24	23	22	21	20	19	16	15	12	11	0
cond		0	0	I	0	0	1	0	S	Rn		Rd		shifter_operand	
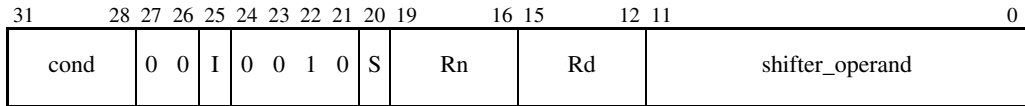
SUB (Subtract) subtracts one value from a second value.

The second value comes from a register. The first value can be either an immediate value or a value from a register, and can be shifted before the subtraction.

SUB can optionally update the condition code flags, based on the result.

Syntax

```
SUB{<cond>}{S}  <Rd>, <Rn>, <shifter_operand>
```

where:

<cond>   Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-7. If <cond> is omitted, the AL (always) condition is used.

S   Sets the S bit (bit[20]) in the instruction to 1 and specifies that the instruction updates the CPSR. If S is omitted, the S bit is set to 0 and the CPSR is not changed by the instruction. Two types of CPSR update can occur when S is specified:

   •   If <Rd> is not R15, the N and Z flags are set according to the result of the subtraction, and the C and V flags are set according to whether the subtraction generated a borrow (unsigned underflow) and a signed overflow, respectively. The rest of the CPSR is unchanged.

   •   If <Rd> is R15, the SPSR of the current mode is copied to the CPSR. This form of the instruction is UNPREDICTABLE if executed in User mode or System mode, because these modes do not have an SPSR.

<Rd>   Specifies the destination register.

<Rn>   Specifies the register that contains the first operand.

<shifter_operand>

               Specifies the second operand. The options for this operand are described in *Addressing Mode 1 - Data-processing operands* on page A5-2, including how each option causes the I bit (bit[25]) and the shifter_operand bits (bits[11:0]) to be set in the instruction.

               If the I bit is 0 and both bit[7] and bit[4] of shifter_operand are 1, the instruction is not SUB. Instead, see *Extending the instruction set* on page A3-35 to determine which instruction it is.

Architecture version

All.

Exceptions

None.

Operation

```
if ConditionPassed(cond) then
    Rd = Rn - shifter_operand
    if S == 1 and Rd == R15 then
        if CurrentModeHasSPSR() then
            CPSR = SPSR
        else UNPREDICTABLE
    else if S == 1 then
        N Flag = Rd[31]
        Z Flag = if Rd == 0 then 1 else 0
        C Flag = NOT BorrowFrom(Rn - shifter_operand)
        V Flag = OverflowFrom(Rn - shifter_operand)
```

Usage

Use SUB to subtract one value from another. To decrement a register value (in Ri) use:

```
    SUB     Ri, Ri, #1
```

SUBS is useful as a loop counter decrement, as the loop branch can test the flags for the appropriate termination condition, without the need for a separate compare instruction:

```
    SUBS    Ri, Ri, #1
```

This both decrements the loop counter in Ri and checks whether it has reached zero.

You can use SUB, with the PC as its destination register and the S bit set, to return from interrupts and various other types of exception. See *Exceptions* on page A2-15 for more details.

Notes

C flag   If S is specified, the C flag is set to:

1   if no borrow occurs

0   if a borrow does occur.

In other words, the C flag is used as a NOT(borrow) flag. This inversion of the borrow condition is used by subsequent instructions: SBC and RSC use the C flag as a NOT(borrow) operand, performing a normal subtraction if C == 1 and subtracting one more than usual if C == 0.

The HS (unsigned higher or same) and LO (unsigned lower) conditions are equivalent to CS (carry set) and CC (carry clear) respectively.

A4.1.110 SUNPK16TO32

31	28	27 26 25 24 23 22 21 20	19 18 17 16	15	12	11 10	9 8	7 6 5 4	3	0
cond		0 1 1 0 1 0 1 1	1 1 1 1	Rd		rotate	SBZ	0 1 1 1	Rm	
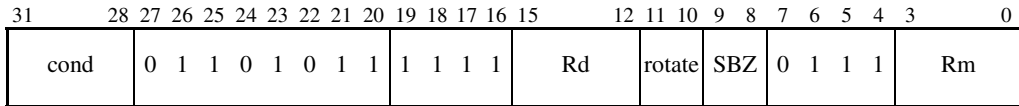
`SUNPK16TO32` (Unpack and Sign extend) extracts a 16-bit value from a register and sign extends it to 32 bits. You can specify a rotation by 0, 8, 16, or 24 bits before extracting the 16-bit value.

Syntax

```
SUNPK16TO32{<cond>}  <Rd>, <Rm>{, <rotation>}
```

where:

`<cond>`	Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-7. If `<cond>` is omitted, the `AL` (always) condition is used.
`<Rd>`	Specifies the destination register.
`<Rm>`	Specifies the register that contains the second operand.
`<rotation>`	This can be any one of:

- `ROR #8`. This is encoded as 0b01 in the rotate field.
- `ROR #16`. This is encoded as 0b10 in the rotate field.
- `ROR #24`. This is encoded as 0b11 in the rotate field.
- Omitted. This is encoded as 0b00 in the rotate field.

───── **Note** ─────

If your assembler accepts shifts by `#0` and treats them as equivalent to no shift or `LSL #0`, then it must accept `ROR #0` here. It is equivalent to omitting `<rotation>`.

─────────────

Architecture version

Version 6 and above.

Exceptions

None.

Operation

```
if ConditionPassed(cond) then
    operand2 = Rm Rotate_Right(8 * rotate)
    Rd[31:0] = SignExtend(operand2[15:0])
```

Usage

Use `SUNPK16TO32` to sign-extend a halfword to a word, for example in instruction sequences acting on **signed short** values in C/C++.

Notes

Use of R15   Specifying R15 for register `<Rd>` or `<Rm>` has UNPREDICTABLE results

A4.1.111 SUNPK8TO16

31	28	27	26	25	24	23	22	21	20	19	18	17	16	15	12	11	10	9	8	7	6	5	4	3	0
cond		0	1	1	0	1	0	0	0	1	1	1	1	Rd		rotate		SBZ		0	1	1	1	Rm	

SUNPK8TO16 extracts two 8-bit values from a register and sign extends them to 16 bits each. You can specify a rotation by 0, 8, 16, or 24 bits before extracting the 8-bit values.

Syntax

SUNPK8TO16{<cond>} <Rd>, <Rm>{, <rotation>}

where:

<cond>   Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-7. If <cond> is omitted, the AL (always) condition is used.

<Rd>   Specifies the destination register.

<Rm>   Specifies the register that contains the second operand.

<rotation>   This can be any one of:

- ROR #8. This is encoded as 0b01 in the rotate field.

- ROR #16. This is encoded as 0b10 in the rotate field.

- ROR #24. This is encoded as 0b11 in the rotate field.

- Omitted. This is encoded as 0b00 in the rotate field.

—— **Note** ——

If your assembler accepts shifts by #0 and treats them as equivalent to no shift or LSL #0, then it must accept ROR #0 here. It is equivalent to omitting <rotation>.

Architecture version

Version 6 and above.

Exceptions

None.

Operation

```
if ConditionPassed(cond) then
    operand2  = Rm Rotate_Right(8 * rotate)
    Rd[15:0]  = SignExtend(operand2[7:0])
    Rd[31:16] = SignExtend(operand2[23:16])
```

Usage

Use `SUNPK8TO16` when you need to keep intermediate values to higher precision while working on arrays of signed byte values. See *UADD8TO16* on page A4-238 for an example of a similar usage.

Notes

Use of R15   Specifying R15 for register `<Rd>` or `<Rm>` has UNPREDICTABLE results

A4.1.112 SUNPK8TO32

31	28	27	26	25	24	23	22	21	20	19	18	17	16	15	12	11	10	9	8	7	6	5	4	3	0
cond		0	1	1	0	1	0	1	0	1	1	1	1	Rd		rotate	SBZ	0 1 1 1				Rm

SUNPK8TO32 extracts an 8-bit value from a register and sign extends it to 32 bits. You can specify a rotation by 0, 8, 16, or 24 bits before extracting the 8-bit value.

Syntax

```
SUNPK8TO32{<cond>}  <Rd>, <Rm>{, <rotation>}
```

where:

<cond>   Is the condition under which the instruction is executed. The conditions are defined
              in *The condition field* on page A3-7. If <cond> is omitted, the AL (always)
              condition is used.

<Rd>   Specifies the destination register.

<Rm>   Specifies the register that contains the second operand.

<rotation>   This can be any one of:

              •   ROR #8. This is encoded as 0b01 in the rotate field.

              •   ROR #16. This is encoded as 0b10 in the rotate field.

              •   ROR #24. This is encoded as 0b11 in the rotate field.

              •   Omitted. This is encoded as 0b00 in the rotate field.

              ———— **Note** ————

              If your assembler accepts shifts by #0 and treats them as equivalent to no
              shift or LSL #0, then it must accept ROR #0 here. It is equivalent to
              omitting <rotation>.

Architecture version

Version 6 and above.

Exceptions

None.

Operation

```
if ConditionPassed(cond) then
    operand2 = Rm Rotate_Right(8 * rotate)
    Rd[31:0] = SignExtend(operand2[7:0])
```

Usage

Use SUNPK8TO32 to sign-extend a byte to a word, for example in instruction sequences acting on **signed char** values in C/C++.

Notes

Use of R15   Specifying R15 for register <Rd> or <Rm> has UNPREDICTABLE results

---

A4.1.113 SWI

31	28	27	26	25	24	23	0

cond	1	1	1	1	immed_24
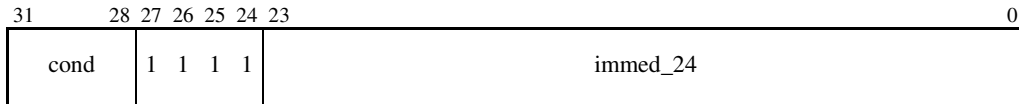
SWI (Software Interrupt) causes a SWI exception (see *Exceptions* on page A2-15).

Syntax

SWI{<cond>}   <immed_24>

where:

<cond>   Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-7. If <cond> is omitted, the AL (always) condition is used.

<immed_24> Is a 24-bit immediate value that is put into bits[23:0] of the instruction. This value is ignored by the ARM processor, but can be used by an operating system SWI exception handler to determine what operating system service is being requested (see *Usage* on page A4-223 below for more details).

Architecture version

All.

Exceptions

Software interrupt.

Operation

```
if ConditionPassed(cond) then
    R14_svc  = address of next instruction after the SWI instruction
    SPSR_svc = CPSR
    CPSR[4:0] = 0b10011             /* Enter Supervisor mode */
    CPSR[5]   = 0                   /* Execute in ARM state */
    /* CPSR[6] is unchanged */
    CPSR[7]   = 1                   /* Disable normal interrupts */
    if high vectors configured then
        PC    = 0xFFFF0008
    else
        PC    = 0x00000008
```

Usage

SWI is used as an operating system service call. The method used to select which operating system service is required is specified by the operating system, and the SWI exception handler for the operating system determines and provides the requested service. Two typical methods are:

*   The 24-bit immediate in the instruction specifies which service is required, and any parameters needed by the selected service are passed in general-purpose registers.

*   The 24-bit immediate in the instruction is ignored, general-purpose register R0 is used to select which service is wanted, and any parameters needed by the selected service are passed in other general-purpose registers.

A4.1.114 SWP

31	28	27 26 25 24 23 22 21 20	19	16	15	12	11	8	7 6 5 4	3	0
cond		0 0 0 1 0 0 0 0	Rn		Rd		SBZ		1 0 0 1	Rm	

SWP (Swap) swaps a word between registers and memory. SWP loads a word from the memory address given by the value of register <Rn>. The value of register <Rm> is then stored to the memory address given by the value of <Rn>, and the original loaded value is written to register <Rd>. If the same register is specified for <Rd> and <Rm>, this instruction swaps the value of the register and the value at the memory address.

Syntax

```
SWP{<cond>}   <Rd>, <Rm>, [<Rn>]
```

where:

<cond>	Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-7. If <cond> is omitted, the AL (always) condition is used.
<Rd>	Specifies the destination register for the instruction.
<Rm>	Contains the value that is stored to memory.
<Rn>	Contains the memory address to load from.

Architecture version

Version 3 and above, plus version 2a.

Exceptions

Data Abort.

Operation

```
if ConditionPassed(cond) then
    if Rn[1:0] == 0b00 then
        temp = Memory[Rn,4]
    else if Rn[1:0] == 0b01 then
        temp = Memory[Rn,4] Rotate_Right 8
    else if Rn[1:0] == 0b10 then
        temp = Memory[Rn,4] Rotate_Right 16
    else /* Rn[1:0] == 0b11 */
        temp = Memory[Rn,4] Rotate_Right 24

    Memory[Rn,4] = Rm    Rd = temp
```

Usage

You can use SWP to implement semaphores.

Notes

Non word-aligned addresses

If the address is not word-aligned, the loaded value is rotated right by 8 times the value of Rn[1:0]. The stored value is not rotated.

Use of R15   If R15 is specified for `<Rd>`, `<Rn>`, or `<Rm>`, the result is UNPREDICTABLE.

Operand restrictions If the same register is specified as `<Rn>` and `<Rm>`, or `<Rn>` and `<Rd>`, the result is UNPREDICTABLE.

Data Abort   If a Data Abort is signaled on either the load access or the store access, the loaded value is not written to `<Rd>`. If a Data Abort is signaled on the load access, the store access does not occur.

Alignment   If an implementation includes a System Control coprocessor (see Chapter B3 *The System Control Coprocessor*), and alignment checking is enabled, an address with bits[1:0] != 0b00 causes an alignment exception.

A4.1.115 SWPB

31	28	27 26 25 24 23 22 21 20	19	16	15	12	11	8	7 6 5 4	3	0
cond		0 0 0 1 0 1 0 0	Rn		Rd		SBZ		1 0 0 1	Rm	

SWPB (Swap Byte) swaps a byte between registers and memory. SWPB loads a byte from the memory address given by the value of register <Rn>. The value of the least significant byte of register <Rm> is stored to the memory address given by <Rn>, the original loaded value is zero-extended to a 32-bit word, and the word is written to register <Rd>. If the same register is specified for <Rd> and <Rm>, this instruction swaps the value of the least significant byte of the register and the byte value at the memory address.

Syntax

```
SWP{<cond>}B  <Rd>, <Rm>, [<Rn>]
```

where:

<cond>   Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-7. If <cond> is omitted, the AL (always) condition is used.

<Rd>   Specifies the destination register for the instruction.

<Rm>   Contains the value that is stored to memory.

<Rn>   Contains the memory address to load from.

Architecture version

Version 3 and above, plus version 2a.

Exceptions

Data Abort.

Operation

```
if ConditionPassed(cond) then
    temp = Memory[Rn,1]
    Memory[Rn,1] = Rm[7:0]
    Rd = temp
```

Usage

You can use SWPB to implement semaphores.

Notes

Use of R15	If R15 is specified for `<Rd>`, `<Rn>`, or `<Rm>`, the result is UNPREDICTABLE.
Operand restrictions	If the same register is specified as `<Rn>` and `<Rm>`, or `<Rn>` and `<Rd>`, the result is UNPREDICTABLE.
Data Abort	If a Data Abort is signaled on either the load access or the store access, the loaded value is not written to `<Rd>`. If a Data Abort is signaled on the load access, the store access does not occur.

A4.1.116 TEQ

31	28	27	26	25	24	23	22	21	20	19	16	15	12	11	0
cond		0	0	I	1	0	0	1	1	Rn		SBZ		shifter_operand	
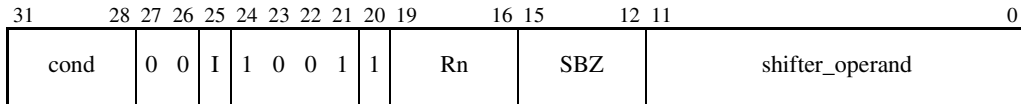
`TEQ` (Test Equivalence) compares a register value with another arithmetic value. The condition flags are updated, based on the result of logically exclusive-ORing the two values, so that subsequent instructions can be conditionally executed.

Syntax

```
TEQ{<cond>}  <Rn>, <shifter_operand>
```

where:

<cond>   Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-7. If `<cond>` is omitted, the `AL` (always) condition is used.

<Rn>   Specifies the register that contains the first operand.

<shifter_operand>

Specifies the second operand. The options for this operand are described in *Addressing Mode 1 - Data-processing operands* on page A5-2, including how each option sets the I bit (bit[25]) and the shifter_operand bits (bits[11:0]) in the instruction.

If the I bit is 0 and both bit[7] and bit[4] of shifter_operand are 1, the instruction is not `TEQ`. Instead, see *Extending the instruction set* on page A3-35 to determine which instruction it is.

Architecture version

All.

Exceptions

None.

Operation

```
if ConditionPassed(cond) then
    alu_out = Rn EOR shifter_operand
    N Flag = alu_out[31]
    Z Flag = if alu_out == 0 then 1 else 0
    C Flag = shifter_carry_out
    V Flag = unaffected
```

Usage

Use TEQ to test if two values are equal, without affecting the V flag (as CMP does). The C flag is also unaffected in many cases. TEQ is also useful for testing whether two values have the same sign. After the comparison, the N flag is the logical Exclusive OR of the sign bits of the two operands.

A4.1.117 TST

31	28	27	26	25	24	23	22	21	20	19	16	15	12	11	0
cond		0	0	I	1	0	0	0	1	Rn		SBZ		shifter_operand	
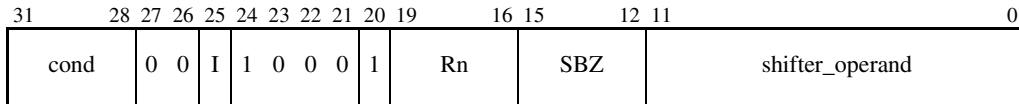
TST (Test) compares a register value with another arithmetic value. The condition flags are updated, based on the result of logically ANDing the two values, so that subsequent instructions can be conditionally executed.

Syntax

TST{<cond>} <Rn>, <shifter_operand>

where:

<cond>   Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-7. If <cond> is omitted, the AL (always) condition is used.

<Rn>   Specifies the register that contains the first operand.

<shifter_operand>

Specifies the second operand. The options for this operand are described in *Addressing Mode 1 - Data-processing operands* on page A5-2, including how each option causes the I bit (bit[25]) and the shifter_operand bits (bits[11:0]) to be set in the instruction.

If the I bit is 0 and both bit[7] and bit[4] of shifter_operand are 1, the instruction is not TST. Instead, see *Extending the instruction set* on page A3-35 to determine which instruction it is.

Architecture version

All.

Exceptions

None.

Operation

```
if ConditionPassed(cond) then
    alu_out = Rn AND shifter_operand
    N Flag = alu_out[31]
    Z Flag = if alu_out == 0 then 1 else 0
    C Flag = shifter_carry_out
    V Flag = unaffected
```

Usage

Use TST to determine whether a particular subset of register bits includes at least one set bit. A very common use for TST is to test whether a single bit is set or clear.

A4.1.118 UADD16

31 28	27 26 25 24 23 22 21 20	19 16	15 12	11 8	7 6 5 4	3 0
cond	0 1 1 0 0 1 0 1	Rn	Rd	SBO	0 0 0 1	Rm

UADD16 (Unsigned Add) performs two 16-bit unsigned integer additions. It sets the GE bits in the CPSR as carry flags for the additions.

Syntax

```
UADD16{<cond>}  <Rd>, <Rn>, <Rm>
```

where:

<cond>   Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-7. If <cond> is omitted, the AL (always) condition is used.

<Rd>   Specifies the destination register.

<Rn>   Specifies the register that contains the first operand.

<Rm>   Specifies the register that contains the second operand.

Architecture version

Version 6 and above.

Exceptions

None.

Operation

```
if ConditionPassed(cond) then
    Rd[15:0]  = Rn[15:0] + Rm[15:0]
    GE[1:0]   = if CarryFrom16(Rn[15:0] + Rm[15:0]) == 1 then 0b11 else 0
    Rd[31:16] = Rn[31:16] + Rm[31:16]
    GE[3:2]   = if CarryFrom16([31:16] + Rm[31:16]) == 1 then 0b11 else 0
```

Usage

UADD16 produces the same result value as SADD16. However, the GE flag values are based on unsigned arithmetic instead of signed arithmetic.

Notes

Use of R15   Specifying R15 for register <Rd>, <Rm>, or <Rn> has UNPREDICTABLE results.

A4.1.119 UADD16TO32

31	28	27	26	25	24	23	22	21	20	19	16	15	12	11	10	9	8	7	6	5	4	3	0
cond		0	1	1	0	1	1	1	1	Rn		Rd		rotate		SBZ		0	1	1	1	Rm	
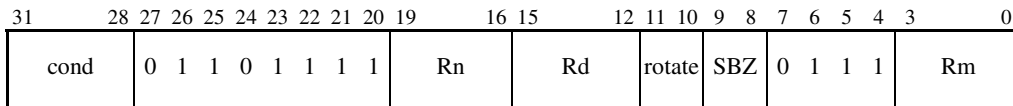
UADD16TO32 extracts a 16-bit value from a register, zero extends it to 32 bits, and adds the result to a value in another register. You can specify a rotation by 0, 8, 16, or 24 bits before extracting the 16-bit value.

Syntax

UADD16TO32{<cond>} <Rd>, <Rn>, <Rm>{, <rotation>}

where:

<cond>   Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-7. If <cond> is omitted, the AL (always) condition is used.

<Rd>   Specifies the destination register.

<Rn>   Specifies the register that contains the first operand.

<Rm>   Specifies the register that contains the second operand.

<rotation>   This can be any one of:

- ROR #8. This is encoded as 0b01 in the rotate field.
- ROR #16. This is encoded as 0b10 in the rotate field.
- ROR #24. This is encoded as 0b11 in the rotate field.
- Omitted. This is encoded as 0b00 in the rotate field.

    ——— **Note** ———

    If your assembler accepts shifts by #0 and treats them as equivalent to no shift or LSL #0, then it must accept ROR #0 here. It is equivalent to omitting <rotation>.

Architecture version

Version 6 and above.

Exceptions

None.

Operation

```
if ConditionPassed(cond) then
    operand2 = (Rm Rotate_Right(8 * rotate)) AND 0x0000ffff
    Rd       = Rn + operand2
```

Usage

You can use `UADD16TO32` to eliminate a separate zero-extension instruction in many instruction sequences that act on **unsigned short** values in C/C++.

Notes

Use of R15   Specifying R15 for register `<Rd>` or `<Rm>` has UNPREDICTABLE results.

——— **Note** ———

Your assembler must fault the use of R15 for register `<Rn>`.

Encoding   If the `<Rn>` field of the instruction contains `0b1111`, the instruction is a `UUNPK16TO32` instruction instead, see *UUNPK16TO32* on page A4-284.

A4.1.120 UADD8

31	28	27 26 25 24 23 22 21 20	19	16	15	12	11	8	7 6 5 4	3	0
cond		0 1 1 0 0 1 0 1	Rn		Rd		SBO		1 0 0 1	Rm	

UADD8 performs four 8-bit unsigned integer additions. It sets the GE bits in the CPSR as carry flags for the additions.

Syntax

```
UADD8{<cond>}  <Rd>, <Rn>, <Rm>
```

where:

<cond>   Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-7. If <cond> is omitted, the AL (always) condition is used.

<Rd>   Specifies the destination register.

<Rn>   Specifies the register that contains the first operand.

<Rm>   Specifies the register that contains the second operand.

Architecture version

Version 6 and above.

Exceptions

None.

Operation

```
if ConditionPassed(cond) then
    Rd[7:0]   = Rn[7:0] + Rm[7:0]
    GE[0]     = CarryFrom8(Rn[7:0] + Rm[7:0])
    Rd[15:8]  = Rn[15:8] + Rm[15:8]
    GE[1]     = CarryFrom8(Rn[15:8] + Rm[15:8])
    Rd[23:16] = Rn[23:16] + Rm[23:16]
    GE[2]     = CarryFrom8(Rn[23:16] + Rm[23:16])
    Rd[31:24] = Rn[31:24] + Rm[31:24]
    GE[3]     = CarryFrom8(Rn[31:24] + Rm[31:24])
```

Usage

UADD8 produces the same result value as SADD8. However, the GE flag values are based on unsigned arithmetic instead of signed arithmetic.

ARM DDI 0100F-02

Notes

Use of R15   Specifying R15 for register `<Rd>`, `<Rm>`, or `<Rn>` has UNPREDICTABLE results.

A4.1.121 UADD8TO16

31 28	27 26 25 24 23 22 21 20	19 16	15 12	11 10	9 8	7 6 5 4	3 0
cond	0 1 1 0 1 1 0 0	Rn	Rd	rotate	SBZ	0 1 1	Rm
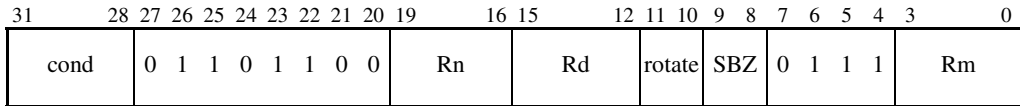
UADD8TO16 extracts two 8-bit values from a register, zero extends them to 16 bits each, and adds the results to the two values from another register. You can specify a rotation by 0, 8, 16, or 24 bits before extracting the 8-bit values.

Syntax

UADD8TO16{<cond>} <Rd>, <Rn>, <Rm>{, <rotation>}

where:

<cond>	Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-7. If <cond> is omitted, the AL (always) condition is used.
<Rd>	Specifies the destination register.
<Rn>	Specifies the register that contains the first operand.
<Rm>	Specifies the register that contains the second operand.
<rotation>	This can be any one of:

- ROR #8. This is encoded as 0b01 in the rotate field.
- ROR #16. This is encoded as 0b10 in the rotate field.
- ROR #24. This is encoded as 0b11 in the rotate field.
- Omitted. This is encoded as 0b00 in the rotate field.

——— **Note** ———

If your assembler accepts shifts by #0 and treats them as equivalent to no shift or LSL #0, then it must accept ROR #0 here. It is equivalent to omitting <rotation>.

Architecture version

Version 6 and above.

Exceptions

None.

Operation

```
if ConditionPassed(cond) then
    operand2  = (Rm Rotate_Right(8 * rotate)) AND 0x00ff00ff
    Rd[15:0]  = Rn[15:0]  + operand2[15:0]
    Rd[31:16] = Rn[31:16] + operand2[23:16]
```

Usage

Use UADD8TO16 to keep intermediate values to higher precision while working on arrays of unsigned byte values.

Notes

Use of R15   Specifying R15 for register <Rd> or <Rm> has UNPREDICTABLE results.

─────── **Note** ───────

Your assembler must fault the use of R15 for register <Rn>.

Encoding   If the <Rn> field of the instruction contains 0b1111, the instruction is an UUNPK8TO16 instruction instead, see *UUNPK8TO16* on page A4-286.

A4.1.122 UADD8TO32

31	28	27	26	25	24	23	22	21	20	19		16	15		12	11	10	9	8	7	6	5	4	3		0
cond		0	1	1	0	1	1	1	0		Rn			Rd		rotate		SBZ		0	1	1			Rm	
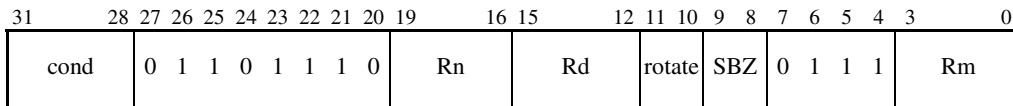
UADD8TO32 extracts an 8-bit value from a register, zero extends it to 32 bits, and adds the result to the value in another register. You can specify a rotation by 0, 8, 16, or 24 bits before extracting the 8-bit value.

Syntax

UADD8TO32{<cond>} <Rd>, <Rn>, <Rm>{, <rotation>}

where:

<cond>   Is the condition under which the instruction is executed. The conditions are defined
                in *The condition field* on page A3-7. If <cond> is omitted, the AL (always)
                condition is used.

<Rd>   Specifies the destination register.

<Rn>   Specifies the register that contains the first operand.

<Rm>   Specifies the register that contains the second operand.

<rotation>   This can be any one of:

                •   ROR #8. This is encoded as 0b01 in the rotate field.

                •   ROR #16. This is encoded as 0b10 in the rotate field.

                •   ROR #24. This is encoded as 0b11 in the rotate field.

                •   Omitted. This is encoded as 0b00 in the rotate field.

                ——— **Note** ———

                If your assembler accepts shifts by #0 and treats them as equivalent to no
                shift or LSL #0, then it must accept ROR #0 here. It is equivalent to
                omitting <rotation>.

Architecture version

Version 6 and above.

Exceptions

None.

ARM DDI 0100F-02

Operation

```
if ConditionPassed(cond) then
    operand2  = (Rm Rotate_Right(8 * rotate)) AND 0x000000ff
    Rd = Rn + operand2
```

Usage

You can use UADD8TO32 to eliminate a separate sign-extension instruction in many instruction sequences that act on **unsigned char** values in C/C++.

Notes

Use of R15   Specifying R15 for register <Rd> or <Rm> has UNPREDICTABLE results.

——— **Note** ———

Your assembler must fault the use of R15 for register <Rn>.

Encoding   If the <Rn> field of the instruction contains 0b1111, the instruction is an UUNPK8TO32 instruction instead, see *UUNPK8TO32* on page A4-288.

A4.1.123 UADDSUBX

31	28	27	26	25	24	23	22	21	20	19	16	15	12	11	8	7	6	5	4	3	0
cond		0	1	1	0	0	1	0	1	Rn		Rd		SBO		0	0	1	1	Rm	

UADDSUBX (Unsigned Add and Subtract with Exchange) performs one 16-bit unsigned integer addition and one 16-bit unsigned integer subtraction. It exchanges the two halfwords of the second operand before it performs the arithmetic. It sets the GE bits in the CPSR according to the results of the addition and subtraction.

Syntax

```
UADDSUBX{<cond>}  <Rd>, <Rn>, <Rm>
```

where:

<cond>	Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-7. If <cond> is omitted, the AL (always) condition is used.
<Rd>	Specifies the destination register.
<Rn>	Specifies the register that contains the first operand.
<Rm>	Specifies the register that contains the second operand.

Architecture version
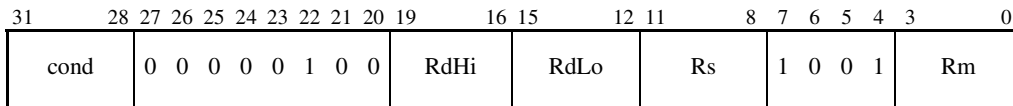
Version 6 and above.

Exceptions

None.

Operation

```
if ConditionPassed(cond) then
    diff     = Rn[15:0] - Rm[31:16]
    Rd[15:0] = diff[15:0]
    GE[1:0]  = if BorrowFrom(Rn[15:0] - Rm[31:16]) then 0 else 0b11
    sum      = Rn[31:16] + Rm[15:0]
    Rd[31:16] = sum[15:0]
    GE[3:2]  = if BorrowFrom(Rn[31:16] + Rm[15:0]) then 0b11 else 0
```

Usage

UADDSUBX produces the same result value as SADDSUBX. However, the GE flag values are based on unsigned arithmetic instead of signed arithmetic.

Notes

Use of R15   Specifying R15 for register <Rd>, <Rm>, or <Rn> has UNPREDICTABLE results.

A4.1.124 UHADD16

31	28	27 26 25 24 23 22 21 20	19	16	15	12	11	8	7 6 5 4	3	0
cond		0 1 1 0 0 1 1 1	Rn		Rd		SBO		0 0 0 1	Rm	

UHADD16 (Unsigned Halving Add) performs two 16-bit unsigned integer additions, and halves the results.

It has no effect on the GE flags.

Syntax

```
UHADD16{<cond>}  <Rd>, <Rn>, <Rm>
```

where:

<cond>   Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-7. If <cond> is omitted, the AL (always) condition is used.

<Rd>   Specifies the destination register.

<Rn>   Specifies the register that contains the first operand.

<Rm>   Specifies the register that contains the second operand.

Architecture version

Version 6 and above.

Exceptions

None.

Operation

```
if ConditionPassed(cond) then
    sum       = Rn[15:0] + Rm[15:0]   /* Unsigned addition */
    Rd[15:0]  = sum[16:1]
    sum       = Rn[31:16] + Rm[31:16] /* Unsigned addition */
    Rd[31:16] = sum[16:1]
```

Usage

Use UHADD16 for similar purposes to UADD16 (see *UADD16* on page A4-232). UHADD16 averages the operands.

Notes

Use of R15   Specifying R15 for register `<Rd>`, `<Rm>`, or `<Rn>` has UNPREDICTABLE results.

A4.1.125 UHADD8

31	28	27 26 25 24 23 22 21 20	19	16	15	12	11	8	7 6 5 4	3	0
cond		0 1 1 0 0 1 1 1	Rn		Rd		SBO		1 0 0 1	Rm	
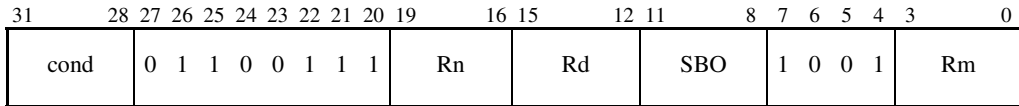
UHADD16 performs four 8-bit unsigned integer additions, and halves the results.

It has no effect on the GE flags.

Syntax

UHADD8{<cond>} <Rd>, <Rn>, <Rm>

where:

<cond>   Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-7. If <cond> is omitted, the AL (always) condition is used.

<Rd>   Specifies the destination register.

<Rn>   Specifies the register that contains the first operand.

<Rm>   Specifies the register that contains the second operand.

Architecture version

Version 6 and above.

Exceptions

None.

Operation

```
if ConditionPassed(cond) then
    sum       = Rn[7:0] + Rm[7:0]          /* Unsigned addition */
    Rd[7:0]   = sum[8:1]
    sum       = Rn[15:8] + Rm[15:8]        /* Unsigned addition */
    Rd[15:8]  = sum[8:1]
    sum       = Rn[23:16] + Rm[23:16]      /* Unsigned addition */
    Rd[23:16] = sum[8:1]
    sum       = Rn[31:24] + Rm[31:24]      /* Unsigned addition */
    Rd[31:24] = sum[8:1]
```

Usage

Use UHADD8 for similar purposes to UADD8 (see *UADD8* on page A4-236). UHADD8 averages the operands.

Notes

Use of R15 Specifying R15 for register `<Rd>`, `<Rm>`, or `<Rn>` has UNPREDICTABLE results.

A4.1.126 UHADDSUBX

31	28	27	26	25	24	23	22	21	20	19	16	15	12	11	8	7	6	5	4	3	0
cond		0	1	1	0	0	1	1	1	Rn		Rd		SBO		0	0	1	1	Rm	

UHADDSUBX (Unsigned Halving Add and Subtract with Exchange) performs one 16-bit unsigned integer addition and one 16-bit unsigned integer subtraction, and halves the results. It exchanges the two halfwords of the second operand before it performs the arithmetic.

It has no effect on the GE flags.

Syntax

UHADDSUBX{<cond>}   <Rd>, <Rn>, <Rm>

where:

<cond>   Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-7. If <cond> is omitted, the AL (always) condition is used.

<Rd>   Specifies the destination register.

<Rn>   Specifies the register that contains the first operand.

<Rm>   Specifies the register that contains the second operand.

Architecture version

Version 6 and above.

Exceptions

None.

Operation

```
if ConditionPassed(cond) then
    diff      = Rn[15:0] - Rm[31:16]  /* Unsigned subtraction */
    Rd[15:0]  = diff[16:1]
    sum       = Rn[31:16] + Rm[15:0]  /* Unsigned addition */
    Rd[31:16] = sum[16:1]
```

Usage

Use UHADDSUBX for similar purposes to UADDSUBX (see *UADDSUBX* on page A4-242). UHADDSUBX halves the results.

Notes

Use of R15   Specifying R15 for register `<Rd>`, `<Rm>`, or `<Rn>` has UNPREDICTABLE results.

A4.1.127 UHSUB16

31	28	27 26 25 24 23 22 21 20	19	16 15	12 11	8	7 6 5 4	3	0
cond		0 1 1 0 0 1 1 1	Rn	Rd	SBO		0 1 1 1	Rm	

`UHSUB16` (Unsigned Halving Subtract) performs two 16-bit unsigned integer subtractions, and halves the results.

It has no effect on the GE flags.

Syntax

```
UHSUB16{<cond>}  <Rd>, <Rn>, <Rm>
```

where:

<cond>   Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-7. If `<cond>` is omitted, the AL (always) condition is used.

<Rd>   Specifies the destination register.

<Rn>   Specifies the register that contains the first operand.

<Rm>   Specifies the register that contains the second operand.

Architecture version

Version 6 and above.

Exceptions

None.

Operation

```
if ConditionPassed(cond) then
    diff      = Rn[15:0] - Rm[15:0]     /* Unsigned subtraction */
    Rd[15:0]  = diff[16:1]
    diff      = Rn[31:16] - Rm[31:16]   /* Unsigned subtraction */
    Rd[31:16] = diff[16:1]
```

Usage

Use `UHSUB16` for similar purposes to `USUB16` (see *USUB16* on page A4-279). `UHSUB16` gives half the difference instead of the full difference.

Notes

Use of R15   Specifying R15 for register `<Rd>`, `<Rm>`, or `<Rn>` has UNPREDICTABLE results.

A4.1.128 UHSUB8

31	28	27 26 25 24 23 22 21 20	19	16	15	12	11	8	7 6 5 4	3	0
cond		0 1 1 0 0 1 1 1	Rn		Rd		SBO		1 1 1 1	Rm	

UHSUB8 performs four 8-bit unsigned integer subtractions, and halves the results.

It has no effect on the GE flags.

Syntax

```
UHSUB8{<cond>}  <Rd>, <Rn>, <Rm>
```

where:

<cond>   Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-7. If <cond> is omitted, the AL (always) condition is used.

<Rd>   Specifies the destination register.

<Rn>   Specifies the register that contains the first operand.

<Rm>   Specifies the register that contains the second operand.

Architecture version

Version 6 and above.

Exceptions

None.

Operation

```
if ConditionPassed(cond) then
    diff      = Rn[7:0] - Rm[7:0]       /* Unsigned subtraction */
    Rd[7:0]   = diff[8:1]
    diff      = Rn[15:8] - Rm[15:8]     /* Unsigned subtraction */
    Rd[15:8]  = diff[8:1]
    diff      = Rn[23:16] - Rm[23:16]   /* Unsigned subtraction */
    Rd[23:16] = diff[8:1]
    diff      = Rn[31:24] - Rm[31:24]   /* Unsigned subtraction */
    Rd[31:24] = diff[8:1]
```

Usage

Use UHSUB8 for similar purposes to USUB8 (see *USUB8* on page A4-280). UHSUB8 gives half the difference instead of the full difference.

ARM DDI 0100F-02

Notes

Use of R15   Specifying R15 for register `<Rd>`, `<Rm>`, or `<Rn>` has UNPREDICTABLE results.

A4.1.129 UHSUBADDX

31	28	27 26 25 24 23 22 21 20	19	16	15	12	11	8	7 6 5 4	3	0
cond		0 1 1 0 0 1 1 1	Rn		Rd		SBO		0 1 0 1	Rm	

UHSUBADDX (Unsigned Halving Subtract and Add with Exchange) performs one 16-bit unsigned integer addition and one 16-bit unsigned integer subtraction, and halves the results. It exchanges the two halfwords of the second operand before it performs the arithmetic.

It has no effect on the GE flags.

Syntax

```
UHSUBADDX{<cond>}  <Rd>, <Rn>, <Rm>
```

where:

<cond>   Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-7. If <cond> is omitted, the AL (always) condition is used.

<Rd>   Specifies the destination register.

<Rn>   Specifies the register that contains the first operand.

<Rm>   Specifies the register that contains the second operand.

Architecture version

Version 6 and above.

Exceptions

None.

Operation

```
if ConditionPassed(cond) then
    sum       = Rn[15:0]  + Rm[31:16]     /* Unsigned addition */
    Rd[15:0]  = sum[16:1]
    diff      = Rn[31:16] - Rm[15:0]      /* Unsigned subtraction */
    Rd[31:16] = diff[16:1]
```

Usage

Use UHSUBADDX for similar purposes to USUBADDX (see *USUBADDX* on page A4-282). UHSUBADDX gives half the difference and the average instead of the full difference and sum.

Notes

Use of R15   Specifying R15 for register `<Rd>`, `<Rm>`, or `<Rn>` has UNPREDICTABLE results.

A4.1.130 UMAAL

31 28	27 26 25 24 23 22 21 20	19 16	15 12	11 8	7 6 5 4	3 0
cond	0 0 0 0 0 1 0 0	RdHi	RdLo	Rs	1 0 0 1	Rm

UMAAL (Unsigned Multiply Accumulate Accumulate Long) multiplies the unsigned value of register `<Rm>` with the unsigned value of register `<Rs>` to produce a 64-bit product. Both the unsigned 32-bit value held in `<RdHi>` and the unsigned 32-bit value held in `<RdLo>` are added to this product, and the sum is written back to `<RdHi>` and `<RdLo>` as a 64-bit value.

Syntax

```
UMAAL{<cond>}  <RdLo>, <RdHi>, <Rm>, <Rs>
```

where:

`<cond>`	Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-7. If `<cond>` is omitted, the AL (always) condition is used.
`<RdLo>`	Supplies one of the 32-bit values to be added to the product of `<Rm>` and `<Rs>`, and is the destination register for the lower 32 bits of the result.
`<RdHi>`	Supplies the other 32-bits value to be added to the product of `<Rm>` and `<Rs>`, and is the destination register for the upper 32 bits of the result.
`<Rm>`	Holds the unsigned value to be multiplied with the value of `<Rs>`.
`<Rs>`	Holds the unsigned value to be multiplied with the value of `<Rm>`.

Architecture version

Version 6 and above.

Exceptions

None.

Operation

```
if ConditionPassed(cond) then
    result = Rm * Rs + RdLo + RdHi  /* Unsigned multiplication and additions */
    RdLo = result[31:0]
    RdHi = result[63:32]
```

Usage

You can use `UMAAL` in multiplying multi-word unsigned integers, in cryptographic applications for example.

Notes

Use of R15   Specifying R15 for register `<RdHi>`, `<RdLo>`, `<Rm>`, or `<Rs>` has
                    UNPREDICTABLE results.

Early termination   If the multiplier implementation supports early termination, it must be implemented
                        on the value of the `<Rs>` operand. The type of early termination used (signed or
                        unsigned) is IMPLEMENTATION DEFINED.

A4.1.131 UMLAL

31	28	27 26 25 24 23 22 21	20	19 16	15 12	11 8	7 6 5 4	3 0
cond		0 0 0 0 1 0 1	S	RdHi	RdLo	Rs	1 0 0 1	Rm

UMLAL (Unsigned Multiply Accumulate Long) multiplies the unsigned value of register <Rm> with the unsigned value of register <Rs> to produce a 64-bit product. This product is added to the 64-bit value held in <RdHi> and <RdLo>, and the sum is written back to <RdHi> and <RdLo>. The condition code flags are optionally updated, based on the result.

Syntax

UMLAL{<cond>}{S}   <RdLo>, <RdHi>, <Rm>, <Rs>

where:

<cond>	Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-7. If <cond> is omitted, the AL (always) condition is used.
S	Causes the S bit (bit[20]) in the instruction to be set to 1 and specifies that the instruction updates the CPSR by setting the N and Z flags according to the result of the multiply-accumulate. If S is omitted, the S bit of the instruction is set to 0 and the entire CPSR is unaffected by the instruction.
<RdLo>	Supplies the lower 32 bits of the value to be added to the product of <Rm> and <Rs>, and is the destination register for the lower 32 bits of the result.
<RdHi>	Supplies the upper 32 bits of the value to be added to the product of <Rm> and <Rs>, and is the destination register for the upper 32 bits of the result.
<Rm>	Holds the signed value to be multiplied with the value of <Rs>.
<Rs>	Holds the signed value to be multiplied with the value of <Rm>.

Architecture version

Versions 3M and 4 and above, excluding xM versions.

Exceptions

None.

ARM DDI 0100F-02

Operation

```
if ConditionPassed(cond) then
    RdLo = (Rm * Rs)[31:0] + RdLo     /* Unsigned multiplication */
    RdHi = (Rm * Rs)[63:32] + RdHi + CarryFrom((Rm * Rs)[31:0] + RdLo)
    if S == 1 then
        N Flag = RdHi[31]
        Z Flag = if (RdHi == 0) and (RdLo == 0) then 1 else 0
        C Flag = unaffected    /* See "C and V flags" note */
        V Flag = unaffected    /* See "C and V flags" note */
```

Usage

UMLAL multiplies unsigned variables to produce a 64-bit result, which is added to the 64-bit value in the two destination general-purpose registers. The result is written back to the two destination general-purpose registers.

Notes

Use of R15   Specifying R15 for register <RdHi>, <RdLo>, <Rm>, or <Rs> has UNPREDICTABLE results.

Operand restriction <RdHi> and <RdLo> must be distinct registers, or the results are UNPREDICTABLE.

Specifying the same register for either <RdHi> and <Rm>, or <RdLo> and <Rm>, was previously described as producing UNPREDICTABLE results. There is no restriction in ARMv6, and it is believed all relevant ARMv4 and ARMv5 implementations do not require this restriction either, because high performance multipliers read all their operands prior to writing back any results.

Early termination   If the multiplier implementation supports early termination, it must be implemented on the value of the <Rs> operand. The type of early termination used (signed or unsigned) is IMPLEMENTATION DEFINED.

C and V flags   UMLALS is defined to leave the C and V flags unchanged in ARM architecture version 5 and above. In earlier versions of the architecture, the values of the C and V flags were UNPREDICTABLE after a UMLALS instruction.

A4.1.132 UMULL
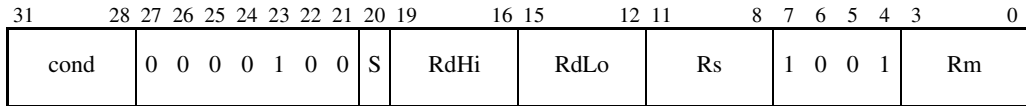
31	28	27 26 25 24 23 22 21	20	19 16	15 12	11 8	7 6 5 4	3 0
cond		0 0 0 0 1 0 0	S	RdHi	RdLo	Rs	1 0 0 1	Rm

UMULL (Unsigned Multiply Long) multiplies the unsigned value of register `<Rm>` with the unsigned value of register `<Rs>` to produce a 64-bit result. The upper 32 bits of the result are stored in `<RdHi>`. The lower 32 bits are stored in `<RdLo>`. The condition code flags are optionally updated, based on the 64-bit result.

Syntax

UMULL{<cond>}{S}   <RdLo>, <RdHi>, <Rm>, <Rs>

where:

`<cond>`	Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-7. If `<cond>` is omitted, the AL (always) condition is used.
`S`	Causes the S bit (bit[20]) in the instruction to be set to 1 and specifies that the instruction updates the CPSR by setting the N and Z flags according to the result of the multiplication. If `S` is omitted, the S bit of the instruction is set to 0 and the entire CPSR is unaffected by the instruction.
`<RdLo>`	Stores the lower 32 bits of the result.
`<RdHi>`	Stores the upper 32 bits of the result.
`<Rm>`	Holds the signed value to be multiplied with the value of `<Rs>`.
`<Rs>`	Holds the signed value to be multiplied with the value of `<Rm>`.

Architecture version

Versions 3M and 4 and above, excluding xM versions.

Exceptions

None.

Operation

```
if ConditionPassed(cond) then
    RdHi = (Rm * Rs)[63:32]    /* Unsigned multiplication */
    RdLo = (Rm * Rs)[31:0]
    if S == 1 then
        N Flag = RdHi[31]
        Z Flag = if (RdHi == 0) and (RdLo == 0) then 1 else 0
        C Flag = unaffected    /* See "C and V flags" note */
        V Flag = unaffected    /* See "C and V flags" note */
```

Usage

UMULL multiplies unsigned variables to produce a 64-bit result in two general-purpose registers.

Notes

Use of R15   Specifying R15 for register <RdHi>, <RdLo>, <Rm>, or <Rs> has UNPREDICTABLE results.

Operand restriction   <RdHi> and <RdLo> must be distinct registers, or the results are UNPREDICTABLE.

Specifying the same register for either <RdHi> and <Rm>, or <RdLo> and <Rm>, was previously described as producing UNPREDICTABLE results. There is no restriction in ARMv6, and it is believed all relevant ARMv4 and ARMv5 implementations do not require this restriction either, because high performance multipliers read all their operands prior to writing back any results.

Early termination   If the multiplier implementation supports early termination, it must be implemented on the value of the <Rs> operand. The type of early termination used (signed or unsigned) is IMPLEMENTATION DEFINED.

C and V flags   UMULLS is defined to leave the C and V flags unchanged in ARM architecture version 5 and above. In earlier versions of the architecture, the values of the C and V flags were UNPREDICTABLE after a UMULLS instruction.

A4.1.133 UQADD16

31	28	27 26 25 24 23 22 21 20	19	16	15	12	11	8	7 6 5 4	3	0
cond		0 1 1 0 0 1 1 0	Rn		Rd		SBO		0 0 0 1	Rm	

UQADD16 (Unsigned Saturating Add) performs two 16-bit integer additions. It saturates the results to the 16-bit unsigned integer range $0 \leq x \leq 2^{16} - 1$.

Syntax

```
UQADD16{<cond>}  <Rd>, <Rn>, <Rm>
```

where:

<cond>   Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-7. If <cond> is omitted, the AL (always) condition is used.

<Rd>   Specifies the destination register.

<Rn>   Specifies the register that contains the first operand.

<Rm>   Specifies the register that contains the second operand.

Architecture version

Version 6 and above.

Exceptions

None.

Operation

```
if ConditionPassed(cond) then
    Rd[15:0]  = UnsignedSat(Rn[15:0]  + Rm[15:0],  16)
    Rd[31:16] = UnsignedSat(Rn[31:16] + Rm[31:16], 16)
```

Usage

Use UQADD16 in similar ways to UADD16, but for unsigned saturated arithmetic. UQADD16 does not set the GE bits for use with SEL. See *UADD16* on page A4-232 for more details.

Notes

Use of R15   Specifying R15 for register <Rd>, <Rm>, or <Rn> has UNPREDICTABLE results.

A4.1.134 UQADD8

31		28	27	26	25	24	23	22	21	20	19		16	15		12	11		8	7	6	5	4	3		0
cond			0	1	1	0	0	1	1	0		Rn			Rd			SBO		1	0	0	1		Rm	

`UQADD8` performs four 8-bit integer additions.

It saturates the results to the 8-bit unsigned integer range $0 \leq x \leq 2^8 - 1$.

Syntax

```
UQADD8{<cond>}  <Rd>, <Rn>, <Rm>
```

where:

<cond>   Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-7. If `<cond>` is omitted, the AL (always) condition is used.

<Rd>   Specifies the destination register.

<Rn>   Specifies the register that contains the first operand.

<Rm>   Specifies the register that contains the second operand.

Architecture version

Version 6 and above.

Exceptions

None.

Operation

```
if ConditionPassed(cond) then
    Rd[7:0]   = UnsignedSat(Rn[7:0]   + Rm[7:0],    8)
    Rd[15:8]  = UnsignedSat(Rn[15:8]  + Rm[15:8],   8)
    Rd[23:16] = UnsignedSat(Rn[23:16] + Rm[23:16],  8)
    Rd[31:24] = UnsignedSat(Rn[31:24] + Rm[31:24],  8)
```

Usage

Use `UQADD8` in similar ways to `UADD8`, but for unsigned saturated arithmetic. `UQADD8` does not set the GE bits for use with `SEL`. See *UADD8* on page A4-236 for more details.

Notes

Use of R15   Specifying R15 for register `<Rd>`, `<Rm>`, or `<Rn>` has UNPREDICTABLE results.

A4.1.135 UQADDSUBX

31	28	27 26 25 24 23 22 21 20	19	16 15	12 11	8 7 6 5 4 3	0
cond		0 1 1 0 0 1 1 0	Rn	Rd	SBO	0 0 1 1	Rm

UQADDSUBX (Unsigned Saturating Add and Subtract with Exchange) performs one 16-bit integer addition and one 16-bit subtraction.

It saturates the results to the 16-bit unsigned integer range $0 \leq x \leq 2^{16} - 1$.

It exchanges the two halfwords of the second operand before it performs the arithmetic.

Syntax

```
UQADDSUBX{<cond>}  <Rd>, <Rn>, <Rm>
```

where:

<cond>   Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-7. If <cond> is omitted, the AL (always) condition is used.

<Rd>   Specifies the destination register.

<Rn>   Specifies the register that contains the first operand.

<Rm>   Specifies the register that contains the second operand.

Architecture version

Version 6 and above.

Exceptions

None.

Operation

```
if ConditionPassed(cond) then
    Rd[15:0]  = UnsignedSat(Rn[15:0]  - Rm[31:16], 16)
    Rd[31:16] = UnsignedSat(Rn[31:16] + Rm[15:0],  16)
```

Use UQADDSUBX in similar ways to UADDSUBX, but for unsigned saturated arithmetic. UQADDSUBX does not set the GE bits for use with SEL. See *UADDSUBX* on page A4-242 for more details.

Notes

Use of R15   Specifying R15 for register <Rd>, <Rm>, or <Rn> has UNPREDICTABLE results.

A4.1.136 UQSUB16

31	28	27	26	25	24	23	22	21	20	19	16	15	12	11	8	7	6	5	4	3	0
cond		0	1	1	0	0	1	1	0	Rn		Rd		SBO		0	1	1	1	Rm	

`UQSUB16` (Unsigned Saturating Subtract) performs two 16-bit subtractions.

It saturates the results to the 16-bit unsigned integer range $0 \leq x \leq 2^{16} - 1$.

Syntax

```
UQSUB16{<cond>}   <Rd>, <Rn>, <Rm>
```

where:

<cond>   Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-7. If `<cond>` is omitted, the AL (always) condition is used.

<Rd>   Specifies the destination register.

<Rn>   Specifies the register that contains the first operand.

<Rm>   Specifies the register that contains the second operand.

Architecture version

Version 6 and above.

Exceptions

None.

Operation

```
if ConditionPassed(cond) then
    Rd[15:0]  = UnsignedSat(Rn[15:0]  - Rm[15:0],  16)
    Rd[31:16] = UnsignedSat(Rn[31:16] - Rm[31:16], 16)
```

Usage

Use `UQSUB16` in similar ways to `USUB16`, but for unsigned saturated arithmetic. `UQSUB16` does not set the GE bits for use with `SEL`. See *SSUB16* on page A4-188 for more details.

Notes

Use of R15   Specifying R15 for register `<Rd>`, `<Rm>`, or `<Rn>` has UNPREDICTABLE results.

ARM DDI 0100F-02

A4.1.137 UQSUB8

31	28	27	26	25	24	23	22	21	20	19	16	15	12	11	8	7	6	5	4	3	0
cond		0	1	1	0	0	1	1	0	Rn		Rd		SBO		1	1	1	1	Rm	
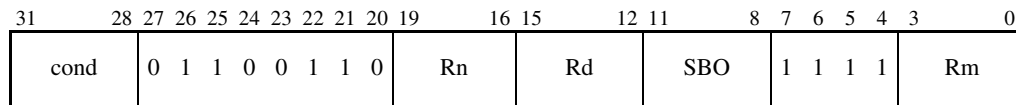
UQSUB8 performs four 8-bit subtractions.

It saturates the results to the 8-bit unsigned integer range $0 \le x \le 2^8 - 1$.

Syntax

```
UQSUB8{<cond>}  <Rd>, <Rn>, <Rm>
```

where:

<cond>   Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-7. If <cond> is omitted, the AL (always) condition is used.

<Rd>   Specifies the destination register.

<Rn>   Specifies the register that contains the first operand.

<Rm>   Specifies the register that contains the second operand.

Architecture version

Version 6 and above.

Exceptions

None.

Operation

```
if ConditionPassed(cond) then
    Rd[7:0]   = UnsignedSat(Rn[7:0]   - Rm[7:0],   8)
    Rd[15:8]  = UnsignedSat(Rn[15:8]  - Rm[15:8],  8)
    Rd[23:16] = UnsignedSat(Rn[23:16] - Rm[23:16], 8)
    Rd[31:24] = UnsignedSat(Rn[31:24] - Rm[31:24], 8)
```

Usage

Use UQSUB8 in similar ways to USUB8, but for unsigned saturated arithmetic. UQSUB8 does not set the GE bits for use with SEL. See *SSUB8* on page A4-190 for more details.

Notes

Use of R15   Specifying R15 for register `<Rd>`, `<Rm>`, or `<Rn>` has UNPREDICTABLE results.

A4.1.138 UQSUBADDX

31	28	27 26 25 24 23 22 21 20	19	16	15	12	11	8	7 6 5 4	3	0
cond		0 1 1 0 0 1 1 0	Rn		Rd		SBO		0 1 0 1	Rm	

UQSUBADDX (Unsigned Saturating Subtract and Add with Exchange) performs one 16-bit integer addition and one 16-bit integer subtraction.

It saturates the results to the 16-bit unsigned integer range $0 \le x \le 2^{16} - 1$.

It exchanges the two halfwords of the second operand before it performs the arithmetic.

Syntax

```
UQSUBADDX{<cond>}  <Rd>, <Rn>, <Rm>
```

where:

<cond>	Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-7. If <cond> is omitted, the AL (always) condition is used.
<Rd>	Specifies the destination register.
<Rn>	Specifies the register that contains the first operand.
<Rm>	Specifies the register that contains the second operand.

Architecture version

Version 6 and above.

Exceptions

None.

Operation

```
if ConditionPassed(cond) then
    Rd[15:0]  = UnsignedSat(Rn[15:0]  + Rm[31:16], 16)
    Rd[31:16] = UnsignedSat(Rn[31:16] - Rm[15:0],  16)
```

Usage

You can use UQSUBADDX in similar ways to USUBADDX, but for unsigned saturated arithmetic. UQSUBADDX does not set the GE bits for use with SEL. See *UADDSUBX* on page A4-242 for more details.

Notes

Use of R15   Specifying R15 for register `<Rd>`, `<Rm>`, or `<Rn>` has UNPREDICTABLE results.

ARM DDI 0100F-02

A4.1.139 USAD8

31 28	27 26 25 24 23 22 21 20	19 16	15 14 13 12	11 8	7 6 5 4	3 0
cond	0 1 1 1 1 0 0 0	Rd	1 1 1 1	Rs	0 0 0 1	Rm

`USAD8` (Unsigned Sum of Absolute Differences) performs four unsigned 8-bit subtractions, and adds the absolute values of the differences together.

Syntax

```
USAD8{<cond>}  <Rd>, <Rm>, <Rs>
```

where:

`<cond>`	Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-7. If `<cond>` is omitted, the AL (always) condition is used.
`<Rd>`	Specifies the destination register.
`<Rm>`	Specifies the register that contains the first operand.
`<Rs>`	Specifies the register that contains the second operand.

Architecture version

Version 6 and above.

Exceptions

None.

Operation

```
if ConditionPassed(cond) then

    if Rm[7:0] < Rs[7:0] then          /* Unsigned comparison */
        diff1 = Rs[7:0] - Rm[7:0]
    else
        diff1 = Rm[7:0] - Rs[7:0]

    if Rm[15:8] < Rs[15:8] then        /* Unsigned comparison */
        diff1 = Rs[15:8] - Rm[15:8]
    else
        diff1 = Rm[15:8] - Rs[15:8]

    if Rm[23:16] < Rs[23:16] then      /* Unsigned comparison */
        diff1 = Rs[23:16] - Rm[23:16]
    else
        diff1 = Rm[23:16] - Rs[23:16]

    if Rm[31:24] < Rs[31:24] then      /* Unsigned comparison */
        diff1 = Rs[31:24] - Rm[31:24]
    else
        diff1 = Rm[31:24] - Rs[31:24]

    Rd = ZeroExtend(diff1) + ZeroExtend(diff2)
                        + ZeroExtend(diff3) + ZeroExtend(diff4]
```

Usage

You can use USAD8 to process the first four bytes in a video motion estimation calculation.

Notes

Use of R15   Specifying R15 for register <Rd>, <Rm>, or <Rs> has UNPREDICTABLE results.

A4.1.140 USADA8

31 28	27 26 25 24 23 22 21 20	19 16	15 12	11 8	7 6 5 4	3 0
cond	0 1 1 1 1 0 0 0	Rd	Rn	Rs	0 0 0 1	Rm

`USADA8` (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.

Syntax

`USADA8{<cond>}  <Rd>, <Rm>, <Rs>, <Rn>`

where:

`<cond>`	Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-7. If `<cond>` is omitted, the `AL` (always) condition is used.
`<Rd>`	Specifies the destination register.
`<Rm>`	Specifies the register that contains the first main operand.
`<Rs>`	Specifies the register that contains the second main operand.
`<Rn>`	Specifies the register that contains the accumulate operand.

Architecture version

Version 6 and above.

Exceptions

None.

Operation

```
if ConditionPassed(cond) then

    if Rm[7:0] < Rs[7:0] then          /* Unsigned comparison */
        diff1 = Rs[7:0] - Rm[7:0]
    else
        diff1 = Rm[7:0] - Rs[7:0]

    if Rm[15:8] < Rs[15:8] then        /* Unsigned comparison */
        diff1 = Rs[15:8] - Rm[15:8]
    else
        diff1 = Rm[15:8] - Rs[15:8]

    if Rm[23:16] < Rs[23:16] then      /* Unsigned comparison */
        diff1 = Rs[23:16] - Rm[23:16]
    else
        diff1 = Rm[23:16] - Rs[23:16]

    if Rm[31:24] < Rs[31:24] then      /* Unsigned comparison */
        diff1 = Rs[31:24] - Rm[31:24]
    else
        diff1 = Rm[31:24] - Rs[31:24]

    Rd = Rn + ZeroExtend(diff1) + ZeroExtend(diff2)
                        + ZeroExtend(diff3) + ZeroExtend(diff4]
```

Usage

You can use USADA8 in video motion estimation calculations.

Notes

Use of R15 Specifying R15 for register <Rd>, <Rm>, or <Rs> has UNPREDICTABLE results.

--- **Note** ---

Your assembler must fault the use of R15 for register <Rn>.

Encoding If the <Rn> field of the instruction contains 0b1111, the instruction is a USAD8 instruction instead, see *USAD8* on page A4-271.

A4.1.141 USAT
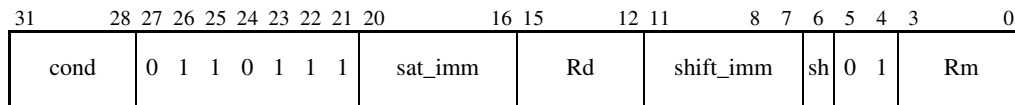
31	28	27 26 25 24 23 22 21	20	16	15	12	11	8	7	6	5	4	3	0
cond		0 1 1 0 1 1 1	sat_imm		Rd		shift_imm		sh	0	1		Rm	

USAT (Unsigned Saturate) saturates a signed value to an unsigned range. You can choose the bit position at which saturation occurs.

You can apply a shift to the value before the saturation occurs.

The Q flag is set if the operation saturates.

Syntax

```
USAT{<cond>}  <Rd>, #<immed>, <Rm>{, <shift>}
```

where:

<cond>   Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-7. If <cond> is omitted, the AL (always) condition is used.

<Rd>   Specifies the destination register.

<immed>   Specifies the bit position for saturation. This lies in the range 0 to 31. It is encoded in the sat_imm field of the instruction.

<Rm>   Specifies the register that contains the value to be saturated.

<shift>   Specifies the optional shift. If present, it must be one of:

- LSL #*N*. *N* must be in the range 0 to 31.
  This is encoded as sh == 0 and shift_imm == *N*.

- ASR #*N*. *N* must be in the range 1 to 32. This is encoded as sh == 1 and either shift_imm == 0 for *N* == 32, or shift_imm == *N* otherwise.

If <shift> is omitted, LSL #0 is used.

Return

The value returned in each half of Rd is:

0   if X is < 0

X   if $0 <= X < 2^n$

$2^n - 1$ if $X > 2^n - 1$

where n is <immed>, and X is the shifted value from Rm.

Architecture version

Version 6 and above.

Exceptions

None.

Operation

```
if ConditionPassed(cond) then
    if shift == 1 then
        if shift_imm == 0 then
            operand = (Rm Artihmetic_Shift_Right 32)[31:0]
        else
            operand = (Rm Artihmetic_Shift_Right shift_imm)[31:0]
    else
        operand = (Rm Logical_Shift_Left shift_imm)[31:0]
    Rd = UnsignedSat(operand, sat_imm)           /* operand treated as signed */
    if UnsignedDoesSat(operand, sat_imm) then
        Q Flag = 1
```

Usage

You can use USAT in various DSP algorithms, such as calculating a pixel color component, that require scaling and saturation of signed data to an unsigned destination.

Notes

Use of R15   Specifying R15 for register <Rd> or <Rm> has UNPREDICTABLE results.

ARM DDI 0100F-02

A4.1.142 USAT16

31	28	27	26	25	24	23	22	21	20	19		16	15		12	11		8	7	6	5	4	3		0
cond		0	1	1	0	1	1	1	0	sat_imm			Rd			SBO			0	0	1	1		Rm	
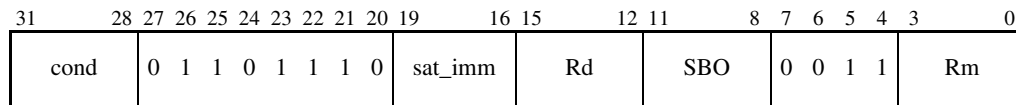
USAT16 saturates two signed 16-bit values to an unsigned range. You can choose the bit position at which saturation occurs.

Syntax

```
USAT16{<cond>}  <Rd>, #<immed>, <Rm>
```

where:

<cond>	Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-7. If <cond> is omitted, the AL (always) condition is used.
<Rd>	Specifies the destination register.
<immed>	Specifies the bit position for saturation. This lies in the range 0 to 15. It is encoded in the sat_imm field of the instruction.
<Rm>	Specifies the register that contains the values to be saturated.

Return

The value returned in each half of Rd is:

0	if X is < 0
X	if $0 <= X < 2^n$
$2^n - 1$	if $X > 2^n - 1$

where n is <immed>, and X is the value from the corresponding half of Rm.

Architecture version

Version 6 and above.

Exceptions

None.

Operation

```
if ConditionPassed(cond) then
    Rd[15:0]  = UnsignedSat(Rm[15:0],  sat_imm) // Rm[15:0]  treated as signed
    Rd[31:16] = UnsignedSat(Rm[31:16], sat_imm) // Rm[31:16] treated as signed
    if UnsignedDoesSat(Rm[15:0], sat_imm)
                                OR UnsignedDoesSat(Rm[31:16], sat_imm) then
        Q Flag = 1
```

Usage

You can use USAT16 in various DSP algorithms, such as calculating a pixel color component, that require saturation of signed data to an unsigned destination.

Notes

Use of R15   Specifying R15 for register <Rd> or <Rm> has UNPREDICTABLE results.

A4.1.143 USUB16

31	28	27	26	25	24	23	22	21	20	19			16	15		12	11		8	7	6	5	4	3		0
cond		0	1	1	0	0	1	0	1		Rn				Rd			SBO		0	1	1	1		Rm	
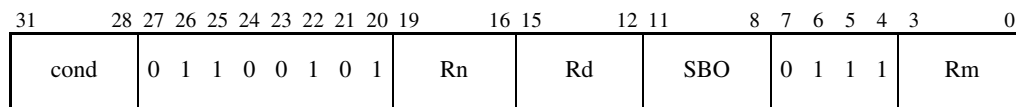
`USUB16` (Unsigned Subtract) performs two 16-bit unsigned integer subtractions. It sets the GE bits in the CPSR as borrow bits for the subtractions.

Syntax

```
USUB16{<cond>}  <Rd>, <Rn>, <Rm>
```

where:

<cond>   Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-7. If <cond> is omitted, the AL (always) condition is used.

<Rd>   Specifies the destination register.

<Rn>   Specifies the register that contains the first operand.

<Rm>   Specifies the register that contains the second operand.

Architecture version

Version 6 and above.

Exceptions

None.

Operation

```
if ConditionPassed(cond) then
    Rd[15:0]  = Rn[15:0] - Rm[15:0]
    GE[1:0]   = if BorrowFrom(Rn[15:0] - Rm[15:0]) then 0 else 0b11
    Rd[31:16] = Rn[31:16] - Rm[31:16]
    GE[3:2]   = if BorrowFrom(Rn[31:16] - Rm[31:160]) then 0 else 0b11
```

Usage

`USUB16` produces the same result as `SSUB16` (see *SSUB16* on page A4-188), but produces GE bit values based on unsigned arithmetic instead of signed arithmetic.

Notes

Use of R15   Specifying R15 for register <Rd>, <Rm>, or <Rn> has UNPREDICTABLE results.

A4.1.144 USUB8

31	28	27 26 25 24 23 22 21 20	19 16	15 12	11 8	7 6 5 4	3 0
cond		0 1 1 0 0 1 0 1	Rn	Rd	SBO	1 1 1 1	Rm

USUB8 performs four 8-bit unsigned integer subtractions. It sets the GE bits in the CPSR as borrow bits for the subtractions.

Syntax

```
USUB8{<cond>}   <Rd>, <Rn>, <Rm>
```

where:

<cond>   Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-7. If <cond> is omitted, the AL (always) condition is used.

<Rd>   Specifies the destination register.

<Rn>   Specifies the register that contains the first operand.

<Rm>   Specifies the register that contains the second operand.

Architecture version

Version 6 and above.

Exceptions

None.

Operation

```
if ConditionPassed(cond) then
    Rd[7:0]    = Rn[7:0] - Rm[7:0]
    GE[0]      = NOT BorrowFrom(Rn[7:0] - Rm[7:0])
    Rd[15:8]   = Rn[15:8] - Rm[15:8]
    GE[1]      = NOT BorrowFrom(Rn[15:8] - Rm[15:8])
    Rd[23:16]  = Rn[23:16] - Rm[23:16]
    GE[2]      = NOT BorrowFrom(Rn[23:16] - Rm[23:16])
    Rd[31:24]  = Rn[31:24] - Rm[31:24]
    GE[3]      = NOT BorrowFrom(Rn[31:24] - Rm[31:24])
```

Usage

USUB8 produces the same result as SSUB8 (see *SSUB8* on page A4-190), but produces GE bit values based on unsigned arithmetic instead of signed arithmetic.

Notes

Use of R15   Specifying R15 for register `<Rd>`, `<Rm>`, or `<Rn>` has UNPREDICTABLE results.

A4.1.145 USUBADDX

31	28	27 26 25 24 23 22 21 20	19	16	15	12	11	8	7 6 5 4	3	0
cond		0 1 1 0 0 1 0 1	Rn		Rd		SBO		0 1 0 1	Rm	

USUBADDX (Unsigned Subtract and Add with Exchange) performs one 16-bit unsigned integer addition and one 16-bit unsigned integer subtraction.

It exchanges the two halfwords of the second operand before it performs the arithmetic.

It sets the GE bits in the CPSR as borrow and carry bits.

Syntax

```
USUBADDX{<cond>}   <Rd>, <Rn>, <Rm>
```

where:

<cond>   Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-7. If <cond> is omitted, the AL (always) condition is used.

<Rd>   Specifies the destination register.

<Rn>   Specifies the register that contains the first operand.

<Rm>   Specifies the register that contains the second operand.

Architecture version

Version 6 and above.

Exceptions

None.

Operation

```
if ConditionPassed(cond) then
    sum       = Rn[15:0] + Rm[31:16]
    Rd[15:0]  = sum[16:1]
    GE[1:0]   = if CarryFrom(Rn[15:0] + Rm[31:16]) then 0b11 else 0
    diff      = Rn[31:16] - Rm[15:0]
    Rd[31:16] = diff[16:1]
    GE[3:2]   = if BorrowFrom(Rn[31:16] - Rm[15:0]) then 0 else 0b11
```

Usage

USUBADDX produces the same result as SSUBADDX (see *SSUBADDX* on page A4-192), but produces GE bit values based on unsigned arithmetic instead of signed arithmetic.

Notes

Use of R15   Specifying R15 for register <Rd>, <Rm>, or <Rn> has UNPREDICTABLE results.

A4.1.146 UUNPK16TO32

31	28	27	26	25	24	23	22	21	20	19	18	17	16	15			12	11	10	9	8	7	6	5	4	3			0
cond		0	1	1	0	1	1	1	1	1	1	1	1		Rd			rotate		SBZ		0	1	1	1		Rm		
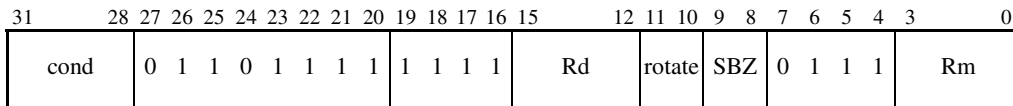
`UUNPK16TO32` (Unsigned Unpack) extracts a 16-bit value from a register and zero extends it to 32 bits. You can specify a rotation by 0, 8, 16, or 24 bits before extracting the 16-bit value.

Syntax

`UUNPK16TO32{<cond>}  <Rd>, <Rm>{, <rotation>}`

where:

`<cond>`	Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-7. If `<cond>` is omitted, the `AL` (always) condition is used.
`<Rd>`	Specifies the destination register.
`<Rm>`	Specifies the register that contains the second operand.
`<rotation>`	This can be any one of:

- `ROR #8`. This is encoded as 0b01 in the rotate field.
- `ROR #16`. This is encoded as 0b10 in the rotate field.
- `ROR #24`. This is encoded as 0b11 in the rotate field.
- Omitted. This is encoded as 0b00 in the rotate field.

———— **Note** ————

If your assembler accepts shifts by `#0` and treats them as equivalent to no shift or `LSL #0`, then it must accept `ROR #0` here. It is equivalent to omitting `<rotation>`.

Architecture version

Version 6 and above.

Exceptions

None.

Operation

```
if ConditionPassed(cond) then
    operand2 = (Rm Rotate_Right(8 * rotate)) AND 0x0000ffff TBD MISSING LINES!
```

Usage

Use `UUNPK16TO32` to zero extend a halfword to a word, for example in instruction sequences acting on **unsigned short** values in C/C++.

Notes

Use of R15   Specifying R15 for register `<Rd>` or `<Rm>` has UNPREDICTABLE results

A4.1.147 UUNPK8TO16

31	28	27	26	25	24	23	22	21	20	19	18	17	16	15	12	11	10	9	8	7	6	5	4	3	0
cond		0 1 1 0 1 1 0 0								1 1 1 1				Rd		rotate		SBZ		0 1 1			Rm	
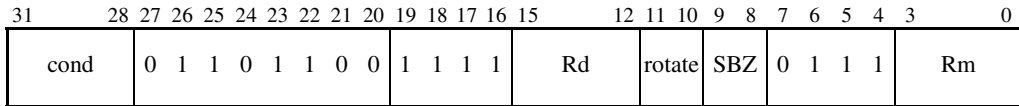
UUNPK8TO16 extracts two 8-bit values from a register and zero extends them to 16 bits each. You can specify a rotation by 0, 8, 16, or 24 bits before extracting the 8-bit values.

Syntax

```
UUNPK8TO16{<cond>}  <Rd>, <Rm>{, <rotation>}
```

where:

<cond>   Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-7. If <cond> is omitted, the AL (always) condition is used.

<Rd>   Specifies the destination register.

<Rm>   Specifies the register that contains the second operand.

<rotation>   This can be any one of:

- ROR #8. This is encoded as 0b01 in the rotate field.

- ROR #16. This is encoded as 0b10 in the rotate field.

- ROR #24. This is encoded as 0b11 in the rotate field.

- Omitted. This is encoded as 0b00 in the rotate field.

  ——— **Note** ———

  If your assembler accepts shifts by #0 and treats them as equivalent to no shift or LSL #0, then it must accept ROR #0 here. It is equivalent to omitting <rotation>.

  ———————————

Architecture version

Version 6 and above.

Exceptions

None.

Operation

```
if ConditionPassed(cond) then
    Rd = (Rm Rotate_Right(8 * rotate)) AND 0x00ff00ff
```

Usage

Use UUNPK8TO16 to zero extend a byte to a halfword, for example in instruction sequences acting on **unsigned char** values in C/C++.

Notes

Use of R15   Specifying R15 for register <Rd> or <Rm> has UNPREDICTABLE results

A4.1.148 UUNPK8TO32

31	28	27	26	25	24	23	22	21	20	19	18	17	16	15	12	11	10	9	8	7	6	5	4	3	0
cond		0	1	1	0	1	1	1	0	1	1	1	1	Rd		rotate		SBZ		0	1	1	1	Rm	

UUNPK8TO32 extracts an 8-bit value from a register and zero extends it to 16 bits. You can specify a rotation by 0, 8, 16, or 24 bits before extracting the 8-bit value.

Syntax

UUNPK8TO32{<cond>} <Rd>, <Rm>{, <rotation>}

where:

<cond>	Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-7. If <cond> is omitted, the AL (always) condition is used.
<Rd>	Specifies the destination register.
<Rm>	Specifies the register that contains the second operand.
<rotation>	This can be any one of:

- ROR #8. This is encoded as 0b01 in the rotate field.
- ROR #16. This is encoded as 0b10 in the rotate field.
- ROR #24. This is encoded as 0b11 in the rotate field.
- Omitted. This is encoded as 0b00 in the rotate field.

---- **Note** ----

If your assembler accepts shifts by #0 and treats them as equivalent to no shift or LSL #0, then it must accept ROR #0 here. It is equivalent to omitting <rotation>.

Architecture version

Version 6 and above.

Exceptions

None.

Operation

```
if ConditionPassed(cond) then
    Rd[31:0] = (Rm Rotate_Right(8 * rotate)) AND 0x000000ff
```

ARM DDI 0100F-02

Usage

Use UUNPK8TO32 to zero extend a byte to a word, for example in instruction sequences acting on **unsigned char** values in C/C++.

Notes

Use of R15   Specifying R15 for register <Rd> or <Rm> has UNPREDICTABLE results

A4.2 ARM instructions and architecture versions

Table A4-2 shows which ARM instructions are present in each current ARM architecture version.

Table A4-2 ARM instructions by architecture version

Instruction	v3, v3M	v4, v4xM	v4T, v4TxM	v5, v5xM, v5T, v5TxM	v5TE, v5TEJ, v5TExP	v6
ADC	Yes	Yes	Yes	Yes	Yes	Yes
ADD	Yes	Yes	Yes	Yes	Yes	Yes
AND	Yes	Yes	Yes	Yes	Yes	Yes
B	Yes	Yes	Yes	Yes	Yes	Yes
BIC	Yes	Yes	Yes	Yes	Yes	Yes
BKPT	No	No	No	Yes	Yes	Yes
BL	Yes	Yes	Yes	Yes	Yes	Yes
BLX (both forms)	No	No	No	Yes	Yes	Yes
BX	No	No	Yes	Yes	Yes	Yes
BXJ	No	No	No	No	Only v5TEJ	Yes
CDP	Yes	Yes	Yes	Yes	Yes	Yes
CDP2	No	No	No	Yes	Yes	Yes
CLZ	No	No	No	Yes	Yes	Yes
CMN	Yes	Yes	Yes	Yes	Yes	Yes
CMP	Yes	Yes	Yes	Yes	Yes	Yes
CPS	No	No	No	No	No	Yes
CPY	No	No	No	No	No	Yes
EOR	Yes	Yes	Yes	Yes	Yes	Yes
LDC	Yes	Yes	Yes	Yes	Yes	Yes
LDC2	No	No	No	Yes	Yes	Yes
LDM (all forms)	Yes	Yes	Yes	Yes	Yes	Yes
LDR	Yes	Yes	Yes	Yes	Yes	Yes

Table A4-2 ARM instructions by architecture version (continued)

Instruction	v3, v3M	v4, v4xM	v4T, v4TxM	v5, v5xM, v5T, v5TxM	v5TE, v5TEJ, v5TExP	v6
LDRB	Yes	Yes	Yes	Yes	Yes	Yes
LDRD	No	No	No	No	Only v5TE, v5TEJ	Yes
LDRBT	Yes	Yes	Yes	Yes	Yes	Yes
LDREX	No	No	No	No	No	Yes
LDRH	No	Yes	Yes	Yes	Yes	Yes
LDRSB	No	Yes	Yes	Yes	Yes	Yes
LDRSH	No	Yes	Yes	Yes	Yes	Yes
LDRT	Yes	Yes	Yes	Yes	Yes	Yes
MCR	Yes	Yes	Yes	Yes	Yes	Yes
MCR2	No	No	No	Yes	Yes	Yes
MCRR	No	No	No	No	Only v5TE, v5TEJ	Yes
MCRR2	No	No	No	No	No	Yes
MLA	Yes	Yes	Yes	Yes	Yes	Yes
MOV	Yes	Yes	Yes	Yes	Yes	Yes
MRC	Yes	Yes	Yes	Yes	Yes	Yes
MRC2	No	No	No	Yes	Yes	Yes
MRRC	No	No	No	No	Only v5TE, v5TEJ	Yes
MRRC2	No	No	No	No	No	Yes
MRS	Yes	Yes	Yes	Yes	Yes	Yes
MSR	Yes	Yes	Yes	Yes	Yes	Yes
MUL	Yes	Yes	Yes	Yes	Yes	Yes
MVN	Yes	Yes	Yes	Yes	Yes	Yes
ORR	Yes	Yes	Yes	Yes	Yes	Yes
PKH (both forms)	No	No	No	No	No	Yes

Table A4-2 ARM instructions by architecture version (continued)

Instruction	v3, v3M	v4, v4xM	v4T, v4TxM	v5, v5xM, v5T, v5TxM	v5TE, v5TEJ, v5TExP	v6
PLD	No	No	No	No	Only v5TE, v5TEJ	Yes
QADD	No	No	No	No	Yes	Yes
QADD16	No	No	No	No	No	Yes
QADD8	No	No	No	No	No	Yes
QADDSUBX	No	No	No	No	No	Yes
QDADD	No	No	No	No	Yes	Yes
QDSUB	No	No	No	No	Yes	Yes
QSUB	No	No	No	No	Yes	Yes
QSUB16	No	No	No	No	No	Yes
QSUB8	No	No	No	No	No	Yes
QSUBADDX	No	No	No	No	No	Yes
REV (all forms)	No	No	No	No	No	Yes
RFE	No	No	No	No	No	Yes
RSB	Yes	Yes	Yes	Yes	Yes	Yes
RSC	Yes	Yes	Yes	Yes	Yes	Yes
SADD (all forms)	No	No	No	No	No	Yes
SBC	Yes	Yes	Yes	Yes	Yes	Yes
SEL	No	No	No	No	No	Yes
SETEND	No	No	No	No	No	Yes
SHADD (all forms)	No	No	No	No	No	Yes
SHSUB (all forms)	No	No	No	No	No	Yes
SMLAD	No	No	No	No	No	Yes
SMLAL	Only v3M	Only v4	Only v4T	Only v5/v5T	Yes	Yes
SMLALD	No	No	No	No	No	Yes

ARM DDI 0100F-02

Table A4-2 ARM instructions by architecture version (continued)

Instruction	v3, v3M	v4, v4xM	v4T, v4TxM	v5, v5xM, v5T, v5TxM	v5TE, v5TEJ, v5TExP	v6
SMLA<x><y>	No	No	No	No	Yes	Yes
SMLAL<x><y>	No	No	No	No	Yes	Yes
SMLAW<y>	No	No	No	No	Yes	Yes
SMLSD	No	No	No	No	No	Yes
SMLSLD	No	No	No	No	No	Yes
SMMLA	No	No	No	No	No	Yes
SMMLS	No	No	No	No	No	Yes
SMMUL	No	No	No	No	No	Yes
SMUAD	No	No	No	No	No	Yes
SMULL	Only v3M	Only v4	Only v4T	Only v5/v5T	Yes	Yes
SMUL<x><y>	No	No	No	No	Yes	Yes
SMULW<y>	No	No	No	No	Yes	Yes
SMUSD	No	No	No	No	No	Yes
SRS	No	No	No	No	No	Yes
SSAT (both forms)	No	No	No	No	No	Yes
SSUB (all forms)	No	No	No	No	No	Yes
STC	Yes	Yes	Yes	Yes	Yes	Yes
STC2	No	No	No	Yes	Yes	Yes
STM (both forms)	Yes	Yes	Yes	Yes	Yes	Yes
STR	Yes	Yes	Yes	Yes	Yes	Yes
STRB	Yes	Yes	Yes	Yes	Yes	Yes
STRBT	Yes	Yes	Yes	Yes	Yes	Yes
STRD	No	No	No	No	Only v5TE, v5TEJ	Yes
STREX	No	No	No	No	No	Yes

Table A4-2 ARM instructions by architecture version (continued)

Instruction	v3, v3M	v4, v4xM	v4T, v4TxM	v5, v5xM, v5T, v5TxM	v5TE, v5TEJ, v5TExP	v6
STRH	No	Yes	Yes	Yes	Yes	Yes
STRT	Yes	Yes	Yes	Yes	Yes	Yes
SUB	Yes	Yes	Yes	Yes	Yes	Yes
SUNPK (all forms)	No	No	No	No	No	Yes
SWI	Yes	Yes	Yes	Yes	Yes	Yes
SWP	Yes	Yes	Yes	Yes	Yes	Yes
SWPB	Yes	Yes	Yes	Yes	Yes	Yes
TEQ	Yes	Yes	Yes	Yes	Yes	Yes
TST	Yes	Yes	Yes	Yes	Yes	Yes
UADD (all forms)	No	No	No	No	No	Yes
UHADD (all forms)	No	No	No	No	No	Yes
UMAAL	No	No	No	No	No	Yes
UMLAL	Only v3M	Only v4	Only v4T	Only v5/v5T	Yes	Yes
UMULL	Only v3M	Only v4	Only v4T	Only v5/v5T	Yes	Yes
UQADD (all forms)	No	No	No	No	No	Yes
UQSUB (all forms)	No	No	No	No	No	Yes
USAD (both forms)	No	No	No	No	No	Yes
USAT (both forms)	No	No	No	No	No	Yes
USUB (all forms)	No	No	No	No	No	Yes
UUNPK (all forms)	No	No	No	No	No	Yes

Chapter A5
ARM Addressing Modes

This chapter describes each of the five addressing modes used with ARM instructions. The chapter contains the following sections:

- *Addressing Mode 1 - Data-processing operands* on page A5-2
- *Addressing Mode 2 - Load and Store Word or Unsigned Byte* on page A5-19
- *Addressing Mode 3 - Miscellaneous Loads and Stores* on page A5-38
- *Addressing Mode 4 - Load and Store Multiple* on page A5-52
- *Addressing Mode 5 - Load and Store Coprocessor* on page A5-60.

A5.1 Addressing Mode 1 - Data-processing operands

There are 11 addressing modes used to calculate the `<shifter_operand>` in an ARM data-processing instruction. The general instruction syntax is:

```
<opcode>{<cond>}{S} <Rd>, <Rn>, <shifter_operand>
```

where `<shifter_operand>` is one of the following 11 options:

1. `#<immediate>`

    See *Data-processing operands - Immediate* on page A5-6.

2. `<Rm>`

    See *Data-processing operands - Register* on page A5-8.

3. `<Rm>, LSL #<shift_imm>`

    See *Data-processing operands - Logical shift left by immediate* on page A5-9.

4. `<Rm>, LSL <Rs>`

    See *Data-processing operands - Logical shift left by register* on page A5-10.

5. `<Rm>, LSR #<shift_imm>`

    See *Data-processing operands - Logical shift right by immediate* on page A5-11.

6. `<Rm>, LSR <Rs>`

    See *Data-processing operands - Logical shift right by register* on page A5-12.

7. `<Rm>, ASR #<shift_imm>`

    See *Data-processing operands - Arithmetic shift right by immediate* on page A5-13.

8. `<Rm>, ASR <Rs>`

    See *Data-processing operands - Arithmetic shift right by register* on page A5-14.

9. `<Rm>, ROR #<shift_imm>`

    See *Data-processing operands - Rotate right by immediate* on page A5-16.

10. `<Rm>, ROR <Rs>`

    See *Data-processing operands - Rotate right by register* on page A5-17.

11. `<Rm>, RRX`

    See *Data-processing operands - Rotate right with extend* on page A5-18.

A5.1.1 Encoding

The following diagrams show the encodings for this addressing mode:

32-bit immediate

31 28	27	26	25	24 21	20	19 16	15 12	11 8	7 0
cond	0	0	1	opcode	S	Rn	Rd	rotate_imm	immed_8

Immediate shifts

31 28	27	26	25	24 21	20	19 16	15 12	11 7	6 5	4	3 0
cond	0	0	0	opcode	S	Rn	Rd	shift_imm	shift	0	Rm

Register shifts

31 28	27	26	25	24 21	20	19 16	15 12	11 8	7	6 5	4	3 0
cond	0	0	0	opcode	S	Rn	Rd	Rs	0	shift	1	Rm
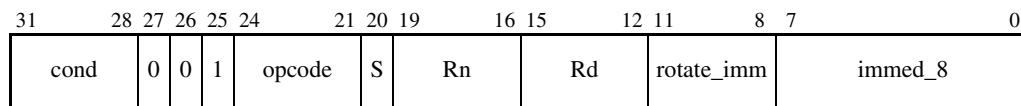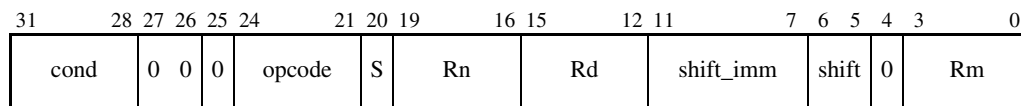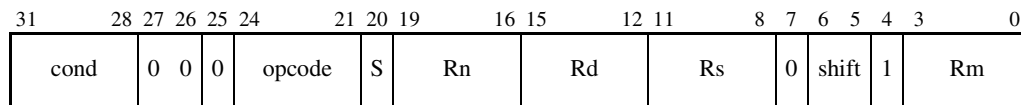
opcode   Specifies the operation of the instruction.

S bit   Indicates that the instruction updates the condition codes.

Rd   Specifies the destination register.

Rn   Specifies the first source operand register.

Bits[11:0] The fields within bits[11:0] are collectively called a *shifter operand*. This is described in *The shifter operand* on page A5-4.

Bit[25]   Is referred to as the I bit, and is used to distinguish between an immediate shifter operand and a register-based shifter operand.

If all three of the following bits have the values shown, the instruction is not a data-processing instruction, but lies in the arithmetic or Load/Store instruction extension space:

```
bit[25]   == 0
bit[4]    == 1
bit[7]    == 1
```

See *Extending the instruction set* on page A3-35 for more information.

A5.1.2   The shifter operand

As well as producing the shifter operand, the shifter produces a carry-out which some instructions write into the Carry Flag. The default register operand (register Rm specified with no shift) uses the form register shift left by immediate, with the immediate set to zero.

The shifter operand takes one of the following three basic formats.

Immediate operand value

An immediate operand value is formed by rotating an 8-bit constant (in a 32-bit word) by an even number of bits (0,2,4,8...26,28,30). Therefore, each instruction contains an 8-bit constant and a 4-bit rotate to be applied to that constant.

Some valid constants are:

```
0xFF,0x104,0xFF0,0xFF00,0xFF000,0xFF000000,0xF000000F
```

Some invalid constants are:

```
0x101,0x102,0xFF1,0xFF04,0xFF003,0xFFFFFFFF,0xF000001F
```

For example:

```
MOV    R0, #0                 ; Move zero to R0
ADD    R3, R3, #1             ; Add one to the value of register 3
CMP    R7, #1000              ; Compare value of R7 with 1000
BIC    R9, R8, #0xFF00        ; Clear bits 8-15 of R8 and store in R9
```

Register operand value

A register operand value is simply the value of a register. The value of the register is used directly as the operand to the data-processing instruction. For example:

```
MOV    R2, R0                 ; Move the value of R0 to R2
ADD    R4, R3, R2             ; Add R2 to R3, store result in R4
CMP    R7, R8                 ; Compare the value of R7 and R8
```

Shifted register operand value

A shifted register operand value is the value of a register, shifted (or rotated) before it is used as the data-processing operand. There are five types of shift:

ASR   Arithmetic shift right

LSL   Logical shift left

LSR   Logical shift right

ROR   Rotate right

RRX   Rotate right with extend.

Copyright © 1996–2003 ARM Limited. All rights reserved.
Confidential - Draft

The number of bits to shift by is specified either as an immediate or as the value of a register. For example:

```
MOV    R2,  R0, LSL  #2           ; Shift R0 left by 2, write to R2, (R2=R0x4)
ADD    R9,  R5, R5,  LSL #3       ; R9 = R5 + R5 x 8 or R9 = R5 x 9
RSB    R9,  R5, R5,  LSL #3       ; R9 = R5 x 8 - R5 or R9 = R5 x 7
SUB    R10, R9, R8,  LSR #4       ; R10 = R9 - R8 / 16
MOV    R12, R4, ROR  R3            ; R12 = R4 rotated right by value of R3
```

A5.1.3 Data-processing operands - Immediate

31 28	27 26 25	24 21	20	19 16	15 12	11 8	7 0
cond	0 0 1	opcode	S	Rn	Rd	rotate_imm	immed_8
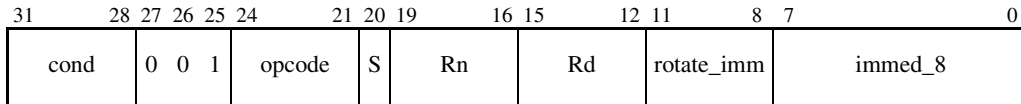
This data-processing operand provides a constant (defined in the instruction) operand to a data-processing instruction.

The `shifter_operand` value is formed by rotating (to the right) an 8-bit immediate value to any even bit position in a 32-bit word. If the rotate immediate is zero, the carry-out from the shifter is the value of the C flag, otherwise, it is set to bit[31] of the value of `<shifter_operand>`.

Syntax

```
#<immediate>
```

where:

`<immediate>` Specifies the immediate constant wanted. It is encoded in the instruction as an 8-bit immediate (immed_8) and a 4-bit immediate (rotate_imm), so that `<immediate>` is equal to the result of rotating immed_8 right by (2 * rotate_imm) bits.

Architecture version

All

Operation

```
shifter_operand = immed_8 Rotate_Right (rotate_imm * 2)
if rotate_imm == 0 then
    shifter_carry_out = C flag
else /* rotate_imm != 0 */
    shifter_carry_out = shifter_operand[31]
```

Notes

Legitimate immediates

Not all 32-bit immediates are legitimate. Only those that can be formed by rotating an 8-bit immediate right by an even amount are valid 32-bit immediates for this format.

Encoding   Some values of `<immediate>` have more than one possible encoding. For example, a value of `0x3F0` could be encoded as:

immed_8 == `0x3F`, rotate_imm == `0xE`

or as:

immed_8 == `0xFC`, rotate_imm == `0xF`

Copyright © 1996–2003 ARM Limited. All rights reserved. ARM DDI 0100F-02
Confidential - Draft

When more than one encoding is available, an assembler must choose the correct one to use, as follows:

- If `<immediate>` lies in the range 0 to `0xFF`, an encoding with rotate_imm == 0 is available. The assembler must choose that encoding. (Choosing another encoding would affect how some instructions set the C flag.)

- Otherwise, it is recommended that the encoding with the smallest value of rotate_imm is chosen. (This choice does not affect instruction functionality.)

For more precise control of the encoding, the instruction fields can be specified directly by using the syntax:

```
#<immed_8>, <rotate_amount>
```

where `<rotate_amount>` = 2 * rotate_imm.

A5.1.4   Data-processing operands - Register

31	28	27 26 25	24	21	20	19	16	15	12	11 10 9 8 7	6 5 4	3	0
cond		0 0 0	opcode		S	Rn		Rd		0 0 0 0 0	0 0 0	Rm	
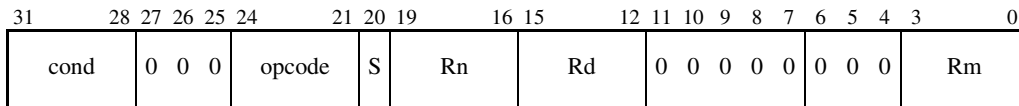
This data-processing operand provides the value of a register directly. The carry-out from the shifter is the C flag.

Syntax

```
<Rm>
```

where:

<Rm>   Specifies the register whose value is the instruction operand.

Architecture version

All

Operation

```
shifter_operand = Rm
shifter_carry_out = C Flag
```

Notes

Encoding   This instruction is encoded as a logical shift left by immediate (see *Data-processing operands - Logical shift left by immediate* on page A5-9) with a shift of zero (shift_imm == 0).

Use of R15   If R15 is specified as register Rm or Rn, the value used is the address of the current instruction plus 8.

A5.1.5 Data-processing operands - Logical shift left by immediate

31		28	27	26	25	24		21	20	19		16	15		12	11		7	6	5	4	3		0
cond			0	0	0	opcode			S	Rn			Rd			shift_imm			0	0	0	Rm		
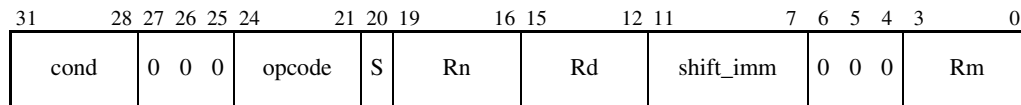
This data-processing operand is used to provide either the value of a register directly (lone register operand, as described in *Data-processing operands - Register* on page A5-8), or the value of a register shifted left (multiplied by a constant power of two).

This instruction operand is the value of register Rm, logically shifted left by an immediate value in the range 0 to 31. Zeros are inserted into the vacated bit positions. The carry-out from the shifter is the last bit shifted out, or the C flag if no shift is specified.

Syntax

```
<Rm>, LSL #<shift_imm>
```

where:

<Rm>   Specifies the register whose value is to be shifted.

LSL   Indicates a logical shift left.

<shift_imm>   Specifies the shift. This is a value between 0 and 31.

Architecture version

All

Operation

```
if shift_imm == 0 then /* Register Operand */
    shifter_operand = Rm
    shifter_carry_out = C Flag
else /* shift_imm > 0 */
    shifter_operand = Rm Logical_Shift_Left shift_imm
    shifter_carry_out = Rm[32 - shift_imm]
```

Notes

Default shift   If the value of <shift_imm> == 0, the operand can be written as just <Rm> (see
                    Data-processing operands - Register on page A5-8).

Use of R15   If R15 is specified as register Rm or Rn, the value used is the address of the current
                    instruction plus 8.

A5.1.6   Data-processing operands - Logical shift left by register
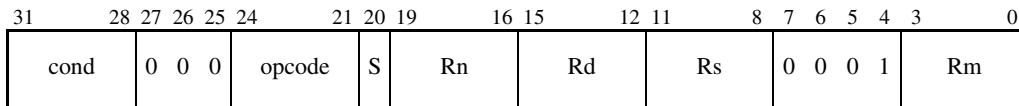
31	28	27 26 25	24	21	20	19	16	15	12	11	8	7 6 5 4	3	0
cond		0 0 0	opcode		S	Rn		Rd		Rs		0 0 0 1	Rm	

This data-processing operand is used to provide the value of a register multiplied by a variable power of two.

This instruction operand is the value of register Rm, logically shifted left by the value in the least significant byte of register Rs. Zeros are inserted into the vacated bit positions. The carry-out from the shifter is the last bit shifted out, which is zero if the shift amount is more than 32, or the C flag if the shift amount is zero.

Syntax

```
<Rm>, LSL <Rs>
```

where:

<Rm>   Specifies the register whose value is to be shifted.

LSL   Indicates a logical shift left.

<Rs>   Is the register containing the value of the shift.

Architecture version

All

Operation

```
if Rs[7:0] == 0 then
    shifter_operand = Rm
    shifter_carry_out = C Flag
else if Rs[7:0] < 32 then
    shifter_operand = Rm Logical_Shift_Left Rs[7:0]
    shifter_carry_out = Rm[32 - Rs[7:0]]
else if Rs[7:0] == 32 then
    shifter_operand = 0
    shifter_carry_out = Rm[0]
else /* Rs[7:0] > 32 */
    shifter_operand = 0
    shifter_carry_out = 0
```

Notes

Use of R15   Specifying R15 as register Rd, register Rm, register Rn, or register Rs has UNPREDICTABLE results.

ARM DDI 0100F-02

A5.1.7   Data-processing operands - Logical shift right by immediate
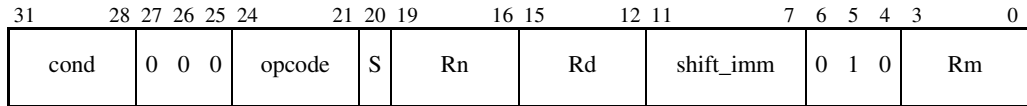
31	28	27 26 25	24	21	20	19	16	15	12	11	7	6 5 4	3	0
cond		0 0 0	opcode		S	Rn		Rd		shift_imm		0 1 0	Rm	

This data-processing operand is used to provide the unsigned value of a register shifted right (divided by a constant power of two).

This instruction operand is the value of register Rm, logically shifted right by an immediate value in the range 1 to 32. Zeros are inserted into the vacated bit positions. The carry-out from the shifter is the last bit shifted out.

Syntax

```
<Rm>, LSR #<shift_imm>
```

where:

`<Rm>`	Specifies the register whose value is to be shifted.
`LSR`	Indicates a logical shift right.
`<shift_imm>`	Specifies the shift. This is an immediate value between 1 and 32. (A shift by 32 is encoded by shift_imm == 0.)

Architecture version

All

Operation

```
if shift_imm == 0 then
    shifter_operand = 0
    shifter_carry_out = Rm[31]
else /* shift_imm > 0 */
    shifter_operand = Rm Logical_Shift_Right shift_imm
    shifter_carry_out = Rm[shift_imm - 1]
```

Notes

Use of R15   If R15 is specified as register Rm or Rn, the value used is the address of the current instruction plus 8.

A5.1.8   Data-processing operands - Logical shift right by register
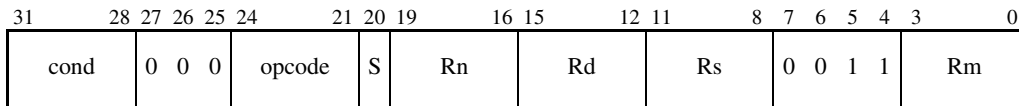
31	28	27 26 25	24	21	20	19	16	15	12	11	8	7 6 5 4	3	0
cond		0 0 0	opcode		S	Rn		Rd		Rs		0 0 1 1	Rm	

This data-processing operand is used to provide the unsigned value of a register shifted right (divided by a variable power of two).

It is produced by the value of register Rm, logically shifted right by the value in the least significant byte of register Rs. Zeros are inserted into the vacated bit positions. The carry-out from the shifter is the last bit shifted out, which is zero if the shift amount is more than 32, or the C flag if the shift amount is zero.

Syntax

```
<Rm>, LSR <Rs>
```

where:

<Rm>   Specifies the register whose value is to be shifted.

LSR   Indicates a logical shift right.

<Rs>   Is the register containing the value of the shift.

Architecture version

All

Operation

```
if Rs[7:0] == 0 then
    shifter_operand = Rm
    shifter_carry_out = C Flag
else if Rs[7:0] < 32 then
    shifter_operand = Rm Logical_Shift_Right Rs[7:0]
    shifter_carry_out = Rm[Rs[7:0] - 1]
else if Rs[7:0] == 32 then
    shifter_operand = 0
    shifter_carry_out = Rm[31]
else /* Rs[7:0] > 32 */
    shifter_operand = 0
    shifter_carry_out = 0
```

Notes

Use of R15   Specifying R15 as register Rd, register Rm, register Rn, or register Rs has UNPREDICTABLE results.

ARM DDI 0100F-02

A5.1.9   Data-processing operands - Arithmetic shift right by immediate
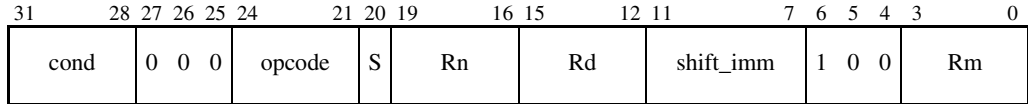
31	28	27	26	25	24			21	20	19				16	15				12	11					7	6	5	4	3			0
cond		0	0	0	opcode				S	Rn					Rd					shift_imm					1	0	0		Rm			

This data-processing operand is used to provide the signed value of a register arithmetically shifted right (divided by a constant power of two).

This instruction operand is the value of register Rm, arithmetically shifted right by an immediate value in the range 1 to 32. The sign bit of Rm (Rm[31]) is inserted into the vacated bit positions. The carry-out from the shifter is the last bit shifted out.

Syntax

```
<Rm>, ASR #<shift_imm>
```

where:

`<Rm>`	Specifies the register whose value is to be shifted.
`ASR`	Indicates an arithmetic shift right.
`<shift_imm>`	Specifies the shift. This is an immediate value between 1 and 32. (A shift by 32 is encoded by shift_imm == 0.)

Architecture version

All

Operation

```
if shift_imm == 0 then
    if Rm[31] == 0 then
        shifter_operand = 0
        shifter_carry_out = Rm[31]
    else /* Rm[31] == 1 */
        shifter_operand = 0xFFFFFFFF
        shifter_carry_out = Rm[31]
else /* shift_imm > 0 */
    shifter_operand = Rm Arithmetic_Shift_Right <shift_imm>
    shifter_carry_out = Rm[shift_imm - 1]
```

Notes

Use of R15   If R15 is specified as register Rm or Rn, the value used is the address of the current instruction plus 8.

A5.1.10 Data-processing operands - Arithmetic shift right by register
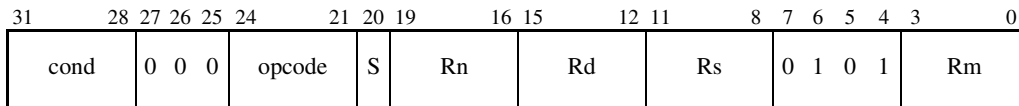
31	28 27 26 25	24	21	20	19	16	15	12	11	8	7 6 5 4	3	0
cond	0 0 0	opcode		S	Rn		Rd		Rs		0 1 0 1	Rm	

This data-processing operand is used to provide the signed value of a register arithmetically shifted right (divided by a variable power of two).

This instruction operand is the value of register Rm arithmetically shifted right by the value in the least significant byte of register Rs. The sign bit of Rm (Rm[31]) is inserted into the vacated bit positions. The carry-out from the shifter is the last bit shifted out, which is the sign bit of Rm if the shift amount is more than 32, or the C flag if the shift amount is zero.

Syntax

```
<Rm>, ASR <Rs>
```

where:

`<Rm>`	Specifies the register whose value is to be shifted.
`ASR`	Indicates an arithmetic shift right.
`<Rs>`	Is the register containing the value of the shift.

Architecture version

All

Operation

```
if Rs[7:0] == 0 then
    shifter_operand = Rm
    shifter_carry_out = C Flag
else if Rs[7:0] < 32 then
    shifter_operand = Rm Arithmetic_Shift_Right Rs[7:0]
    shifter_carry_out = Rm[Rs[7:0] - 1]
else /* Rs[7:0] >= 32 */
    if Rm[31] == 0 then
        shifter_operand = 0
        shifter_carry_out = Rm[31]
    else /* Rm[31] == 1 */
        shifter_operand = 0xFFFFFFFF
        shifter_carry_out = Rm[31]
```

Notes

Use of R15   Specifying R15 as register Rd, register Rm, register Rn, or register Rs has UNPREDICTABLE results.

A5.1.11 Data-processing operands - Rotate right by immediate

31	28	27	26	25	24		21	20	19		16	15		12	11		7	6	5	4	3		0

cond	0 0 0	opcode	S	Rn	Rd	shift_imm	1 1 0	Rm
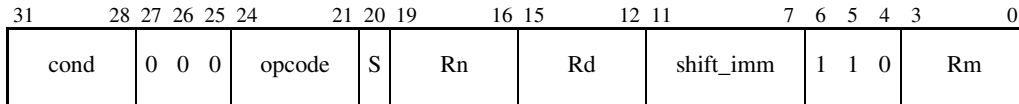
This data-processing operand is used to provide the value of a register rotated by a constant value.

This instruction operand is the value of register Rm rotated right by an immediate value in the range 1 to 31. As bits are rotated off the right end, they are inserted into the vacated bit positions on the left. The carry-out from the shifter is the last bit rotated off the right end.

Syntax

```
<Rm>, ROR #<shift_imm>
```

where:

<Rm>   Specifies the register whose value is to be rotated.

ROR   Indicates a rotate right.

<shift_imm>   Specifies the rotation. This is an immediate value between 1 and 31. When
                shift_imm == 0, an RRX operation (rotate right with extend) is performed. This is
                described in *Data-processing operands - Rotate right with extend* on page A5-18.

Architecture version

All

Operation

```
if shift_imm == 0 then
    See "Data-processing operands - Rotate right with extend" on page A5-17
else /* shift_imm > 0 */
    shifter_operand = Rm Rotate_Right shift_imm
    shifter_carry_out = Rm[shift_imm - 1]
```

Notes

Use of R15   If R15 is specified as register Rm or Rn, the value used is the address of the current
                  instruction plus 8.

A5.1.12 Data-processing operands - Rotate right by register

31	28	27 26 25	24	21	20	19	16	15	12	11	8	7 6 5 4	3	0
cond		0 0 0	opcode		S	Rn		Rd		Rs		0 1 1 1	Rm	

This data-processing operand is used to provide the value of a register rotated by a variable value.

This instruction operand is produced by the value of register Rm rotated right by the value in the least significant byte of register Rs. As bits are rotated off the right end, they are inserted into the vacated bit positions on the left. The carry-out from the shifter is the last bit rotated off the right end, or the C flag if the shift amount is zero.

Syntax

```
<Rm>, ROR <Rs>
```

where:

<Rm>   Specifies the register whose value is to be rotated.

ROR   Indicates a rotate right.

<Rs>   Is the register containing the value of the rotation.

Architecture version

All

Operation

```
if Rs[7:0] == 0 then
    shifter_operand = Rm
    shifter_carry_out = C Flag
else if Rs[4:0] == 0 then
    shifter_operand = Rm
    shifter_carry_out = Rm[31]
else /* Rs[4:0] > 0 */
    shifter_operand = Rm Rotate_Right Rs[4:0]
    shifter_carry_out = Rm[Rs[4:0] - 1]
```

Notes

Use of R15   Specifying R15 as register Rd, register Rm, register Rn, or register Rs has UNPREDICTABLE results.

A5.1.13 Data-processing operands - Rotate right with extend

31	28	27 26 25	24	21	20	19	16	15	12	11 10 9 8 7 6 5 4	3	0
cond		0 0 0	opcode		S	Rn		Rd		0 0 0 0 0 1 1 0	Rm	
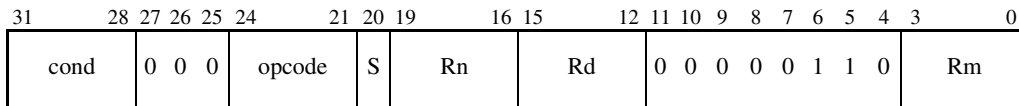
This data-processing operand can be used to perform a 33-bit rotate right using the Carry Flag as the 33rd bit.

This instruction operand is the value of register Rm shifted right by one bit, with the Carry Flag replacing the vacated bit position. The carry-out from the shifter is the bit shifted off the right end.

Syntax

```
<Rm>, RRX
```

where:

<Rm>   Specifies the register whose value is shifted right by one bit.

RRX   Indicates a rotate right with extend.

Architecture version

All

Operation

```
shifter_operand = (C Flag Logical_Shift_Left 31) OR (Rm Logical_Shift_Right 1)
shifter_carry_out = Rm[0]
```

Notes

Encoding   The instruction encoding is in the space that would be used for ROR #0.

Use of R15   If R15 is specified as register Rm or Rn, the value used is the address of the current instruction plus 8.

ADC instruction   A rotate left with extend can be performed with an ADC instruction.

A5.2   **Addressing Mode 2 - Load and Store Word or Unsigned Byte**

There are nine addressing modes used to calculate the address for a Load and Store Word or Unsigned Byte instruction. The general instruction syntax is:

```
LDR|STR{<cond>}{B}{T}  <Rd>, <addressing_mode>
```

where `<addressing_mode>` is one of the nine options listed below.

All nine of the following options are available for `LDR`, `LDRB`, `STR` and `STRB`. For `LDRBT`, `LDRT`, `STRBT` and `STRBT`, only the *post-indexed* options (the last three in the list) are available. For the `PLD` instruction described in *PLD* on page A4-89, only the *offset* options (the first three in the list) are available.

1.   `[<Rn>, #+/-<offset_12>]`

      See *Load and Store Word or Unsigned Byte - Immediate offset* on page A5-21.

2.   `[<Rn>, +/-<Rm>]`

      See *Load and Store Word or Unsigned Byte - Register offset* on page A5-22.

3.   `[<Rn>, +/-<Rm>, <shift> #<shift_imm>]`

      See *Load and Store Word or Unsigned Byte - Scaled register offset* on page A5-23.

4.   `[<Rn>, #+/-<offset_12>]!`

      See *Load and Store Word or Unsigned Byte - Immediate pre-indexed* on page A5-25.

5.   `[<Rn>, +/-<Rm>]!`

      See *Load and Store Word or Unsigned Byte - Register pre-indexed* on page A5-27.

6.   `[<Rn>, +/-<Rm>, <shift> #<shift_imm>]!`

      See *Load and Store Word or Unsigned Byte - Scaled register pre-indexed* on page A5-29.

7.   `[<Rn>], #+/-<offset_12>`

      See *Load and Store Word or Unsigned Byte - Immediate post-indexed* on page A5-31.

8.   `[<Rn>], +/-<Rm>`

      See *Load and Store Word or Unsigned Byte - Register post-indexed* on page A5-33.

9.   `[<Rn>], +/-<Rm>, <shift> #<shift_imm>`

      See *Load and Store Word or Unsigned Byte - Scaled register post-indexed* on page A5-35.

A5.2.1 Encoding

The following three diagrams show the encodings for this addressing mode:
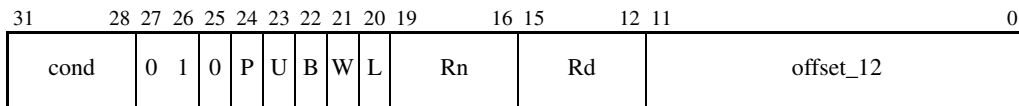
Immediate offset/index

31 28	27 26 25	24	23	22	21	20	19 16	15 12	11 0
cond	0 1 0	P	U	B	W	L	Rn	Rd	offset_12

Register offset/index

31 28	27 26 25	24	23	22	21	20	19 16	15 12	11 10 9 8 7 6 5 4	3 0
cond	0 1 1	P	U	B	W	L	Rn	Rd	0 0 0 0 0 0 0 0	Rm

Scaled register offset/index

31 28	27 26 25	24	23	22	21	20	19 16	15 12	11 7	6 5	4	3 0
cond	0 1 1	P	U	B	W	L	Rn	Rd	shift_imm	shift	0	Rm

The P bit Has two meanings:

P == 0 Indicates the use of *post-indexed addressing*. The base register value is used for the memory address, and the offset is then applied to the base register value and written back to the base register.

P == 1 Indicates the use of *offset addressing* or *pre-indexed addressing* (the W bit determines which). The memory address is generated by applying the offset to the base register value.

The U bit Indicates whether the offset is added to the base (U == 1) or is subtracted from the base (U == 0).

The B bit Distinguishes between an unsigned byte (B == 1) and a word (B == 0) access.

The W bit Has two meanings:

P == 0 If W == 0, the instruction is LDR, LDRB, STR or STRB and a normal memory access is performed. If W == 1, the instruction is LDRBT, LDRT, STRBT or STRT and an unprivileged (User mode) memory access is performed.

P == 1 If W == 0, the base register is not updated (offset addressing). If W == 1, the calculated memory address is written back to the base register (pre-indexed addressing).

The L bit Distinguishes between a Load (L == 1) and a Store (L == 0).

A5.2.2   Load and Store Word or Unsigned Byte - Immediate offset
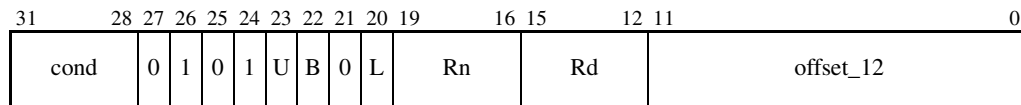
31 28	27	26	25	24	23	22	21	20	19 16	15 12	11 0
cond	0	1	0	1	U	B	0	L	Rn	Rd	offset_12

This addressing mode calculates an address by adding or subtracting the value of an immediate offset to or from the value of the base register Rn.

Syntax

```
[<Rn>, #+/-<offset_12>]
```

where:

<Rn>   Specifies the register containing the base address.

<offset_12>   Specifies the immediate offset used with the value of Rn to form the address.

Architecture version

All

Operation

```
if U == 1 then
    address = Rn + offset_12
else /* U == 0 */
    address = Rn - offset_12
```

Usage

This addressing mode is useful for accessing structure (record) fields, and accessing parameters and local variables in a stack frame. With an offset of zero, the address produced is the unaltered value of the base register Rn.

Notes

Offset of zero The syntax [<Rn>] is treated as an abbreviation for [<Rn>, #0], unless the instruction is one that only allows post-indexed addressing modes (LDRBT, LDRT, STRBT or STRT).

The B bit   This bit distinguishes between an unsigned byte (B==1) and a word (B==0) access.

The L bit   This bit distinguishes between a Load (L==1) and a Store (L==0) instruction.

Use of R15 If R15 is specified as register Rn, the value used is the address of the instruction plus 8.

A5.2.3   Load and Store Word or Unsigned Byte - Register offset
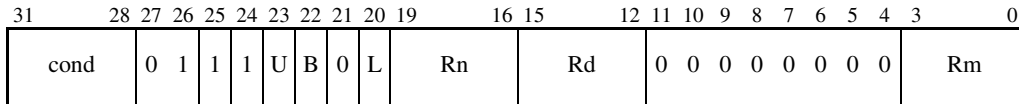
31	28	27	26	25	24	23	22	21	20	19	16	15	12	11	10	9	8	7	6	5	4	3	0
cond		0	1	1	1	U	B	0	L	Rn		Rd		0	0	0	0	0	0	0	0	Rm	

This addressing mode calculates an address by adding or subtracting the value of the index register Rm to or from the value of the base register Rn.

Syntax

```
[<Rn>, +/-<Rm>]
```

where:

<Rn>   Specifies the register containing the base address.

<Rm>   Specifies the register containing the value to add to or subtract from Rn.

Architecture version

All

Operation

```
if U == 1 then
    address = Rn + Rm
else /* U == 0 */
    address = Rn - Rm
```

Usage

This addressing mode is used for pointer plus offset arithmetic, and accessing a single element of an array of bytes.

Notes

Encoding   This addressing mode is encoded as an LSL scaled register offset, scaled by zero.

The B bit   This bit distinguishes between an unsigned byte (B==1) and a word (B==0) access.

The L bit   This bit distinguishes between a Load (L==1) and a Store (L==0) instruction.

Use of R15   If R15 is specified as register Rn, the value used is the address of the instruction plus 8. Specifying R15 as register Rm has UNPREDICTABLE results.

ARM DDI 0100F-02

A5.2.4   Load and Store Word or Unsigned Byte - Scaled register offset
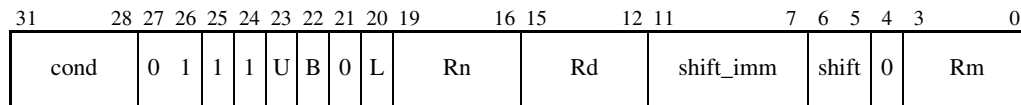
31	28	27	26	25	24	23	22	21	20	19	16	15	12	11	7	6	5	4	3	0
cond		0	1	1	1	U	B	0	L	Rn		Rd		shift_imm		shift		0	Rm	

These five addressing modes calculate an address by adding or subtracting the shifted or rotated value of the index register Rm to or from the value of the base register Rn.

Syntax

One of:

```
[<Rn>, +/-<Rm>, LSL #<shift_imm>]
[<Rn>, +/-<Rm>, LSR #<shift_imm>]
[<Rn>, +/-<Rm>, ASR #<shift_imm>]
[<Rn>, +/-<Rm>, ROR #<shift_imm>]
[<Rn>, +/-<Rm>, RRX]
```

where:

`<Rn>`	Specifies the register containing the base address.
`<Rm>`	Specifies the register containing the offset to add to or subtract from Rn.
`LSL`	Specifies a logical shift left.
`LSR`	Specifies a logical shift right.
`ASR`	Specifies an arithmetic shift right.
`ROR`	Specifies a rotate right.
`RRX`	Specifies a rotate right with extend.
`<shift_imm>`	Specifies the shift or rotation.

	`LSL`	0 to 31, encoded directly in the shift_imm field.
	`LSR`	1 to 32. A shift amount of 32 is encoded as shift_imm == 0. Other shift amounts are encoded directly.
	`ASR`	1 to 32. A shift amount of 32 is encoded as shift_imm == 0. Other shift amounts are encoded directly.
	`ROR`	1 to 31, encoded directly in the shift_imm field. (The shift_imm == 0 encoding is used to specify the `RRX` option.)

Architecture version

All

Operation

```
case shift of
    0b00 /* LSL */
        index = Rm Logical_Shift_Left shift_imm
    0b01 /* LSR */
        if shift_imm == 0 then /* LSR #32 */
            index = 0
        else
            index = Rm Logical_Shift_Right shift_imm
    0b10 /* ASR */
        if shift_imm == 0 then /* ASR #32 */
            if Rm[31] == 1 then
                index = 0xFFFFFFFF
            else
                index = 0
        else
            index = Rm Arithmetic_Shift_Right shift_imm
    0b11 /* ROR or RRX */
        if shift_imm == 0 then /* RRX */
            index = (C Flag Logical_Shift_Left 31) OR
                    (Rm Logical_Shift_Right 1)
        else /* ROR */
            index = Rm Rotate_Right shift_imm
endcase
if U == 1 then
    address = Rn + index
else /* U == 0 */
    address = Rn - index
```

Usage

These addressing modes are used for accessing a single element of an array of values larger than a byte.

Notes

The B bit   This bit distinguishes between an unsigned byte (B==1) and a word (B==0) access.

The L bit   This bit distinguishes between a Load (L==1) and a Store (L==0) instruction.

Use of R15   If R15 is specified as register Rn, the value used is the address of the instruction plus 8. Specifying R15 as register Rm has UNPREDICTABLE results.

A5.2.5   Load and Store Word or Unsigned Byte - Immediate pre-indexed

31	28	27	26	25	24	23	22	21	20	19	16	15	12	11	0
cond		0	1	0	1	U	B	1	L	Rn		Rd		offset_12	

This addressing mode calculates an address by adding or subtracting the value of an immediate offset to or from the value of the base register Rn.

If the condition specified in the instruction matches the condition code status, the calculated address is written back to the base register Rn. The conditions are defined in *The condition field* on page A3-7.

Syntax

```
[<Rn>, #+/-<offset_12>]!
```

where:

`<Rn>`	Specifies the register containing the base address.
`<offset_12>`	Specifies the immediate offset used with the value of Rn to form the address.
`!`	Sets the W bit, causing base register update.

Architecture version

All

Operation

```
if U == 1 then
    address = Rn + offset_12
else /* if U == 0 */
    address = Rn - offset_12
if ConditionPassed(cond) then
    Rn = address
```

Usage

This addressing mode is used for pointer access to arrays with automatic update of the pointer value.

Notes

Offset of zero The syntax [<Rn>] must never be treated as an abbreviation for [<Rn>, #0]!.

The B bit   This bit distinguishes between an unsigned byte (B==1) and a word (B==0) access.

The L bit   This bit distinguishes between a Load (L==1) and a Store (L==0) instruction.

Use of R15   Specifying R15 as register Rn has UNPREDICTABLE results.

A5.2.6   Load and Store Word or Unsigned Byte - Register pre-indexed

31 28	27 26 25 24	23	22	21	20	19 16	15 12	11 10 9 8 7 6 5 4	3 0
cond	0 1 1 1	U	B	1	L	Rn	Rd	0 0 0 0 0 0 0 0	Rm

This addressing mode calculates an address by adding or subtracting the value of an index register Rm to or from the value of the base register Rn.

If the condition specified in the instruction matches the condition code status, the calculated address is written back to the base register Rn. The conditions are defined in *The condition field* on page A3-7.

Syntax

```
[<Rn>, +/-<Rm>]!
```

where:

<Rn>	Specifies the register containing the base address.
<Rm>	Specifies the register containing the offset to add to or subtract from Rn.
!	Sets the W bit, causing base register update.

Architecture version

All

Operation

```
if U == 1 then
    address = Rn + Rm
else /* U == 0 */
    address = Rn - Rm
if ConditionPassed(cond) then
    Rn = address
```

Notes

Encoding	This addressing mode is encoded as an LSL scaled register offset, scaled by zero.
The B bit	This bit distinguishes between an unsigned byte (B==1) and a word (B==0) access.
The L bit	This bit distinguishes between a Load (L==1) and a Store (L==0) instruction.
Use of R15	Specifying R15 as register Rm or Rn has UNPREDICTABLE results.

Operand restriction   There are no operand restrictions in ARM architecture version 6 and above. In earlier versions of the architecture, if the same register is specified for Rn and Rm, the result is UNPREDICTABLE.

A5.2.7   Load and Store Word or Unsigned Byte - Scaled register pre-indexed

31	28	27	26	25	24	23	22	21	20	19	16	15	12	11	7	6	5	4	3	0
cond		0	1	1	1	U	B	1	L	Rn		Rd		shift_imm		shift		0	Rm	

These five addressing modes calculate an address by adding or subtracting the shifted or rotated value of the index register Rm to or from the value of the base register Rn.

If the condition specified in the instruction matches the condition code status, the calculated address is written back to the base register Rn. The conditions are defined in *The condition field* on page A3-7.

Syntax

One of:

```
[<Rn>, +/-<Rm>, LSL #<shift_imm>]!
[<Rn>, +/-<Rm>, LSR #<shift_imm>]!
[<Rn>, +/-<Rm>, ASR #<shift_imm>]!
[<Rn>, +/-<Rm>, ROR #<shift_imm>]!
[<Rn>, +/-<Rm>, RRX]!
```

where:

`<Rn>`	Specifies the register containing the base address.
`<Rm>`	Specifies the register containing the offset to add to or subtract from Rn.
`LSL`	Specifies a logical shift left.
`LSR`	Specifies a logical shift right.
`ASR`	Specifies an arithmetic shift right.
`ROR`	Specifies a rotate right.
`RRX`	Specifies a rotate right with extend.
`<shift_imm>`	Specifies the shift or rotation.

	`LSL`	0 to 31, encoded directly in the shift_imm field.
	`LSR`	1 to 32. A shift amount of 32 is encoded as shift_imm == 0. Other shift amounts are encoded directly.
	`ASR`	1 to 32. A shift amount of 32 is encoded as shift_imm == 0. Other shift amounts are encoded directly.
	`ROR`	1 to 31, encoded directly in the shift_imm field. (The shift_imm == 0 encoding is used to specify the `RRX` option.)
`!`		Sets the W bit, causing base register update.

Copyright © 1996–2003 ARM Limited. All rights reserved.
Confidential - Draft

Architecture version

All

Operation

```
case shift of
    0b00 /* LSL */
        index = Rm Logical_Shift_Left shift_imm
    0b01 /* LSR */
        if shift_imm == 0 then /* LSR #32 */
            index = 0
        else
            index = Rm Logical_Shift_Right shift_imm
    0b10 /* ASR */
        if shift_imm == 0 then /* ASR #32 */
            if Rm[31] == 1 then
                index = 0xFFFFFFFF
            else
                index = 0
        else
            index = Rm Arithmetic_Shift_Right shift_imm
    0b11 /* ROR or RRX */
        if shift_imm == 0 then /* RRX */
            index = (C Flag Logical_Shift_Left 31) OR
                (Rm Logical_Shift_Right 1)
        else /* ROR */
            index = Rm Rotate_Right shift_imm
endcase
if U == 1 then
    address = Rn + index
else /* U == 0 */
    address = Rn - index
if ConditionPassed(cond) then
    Rn = address
```

Notes

The B bit   This bit distinguishes between an unsigned byte (B==1) and a word (B==0) access.

The L bit   This bit distinguishes between a Load (L==1) and a Store (L==0) instruction.

Use of R15   Specifying R15 as register Rm or Rn has UNPREDICTABLE results.

Operand restriction There are no operand restrictions in ARM architecture version 6 and above. In earlier versions of the architecture, if the same register is specified for Rn and Rm, the result is UNPREDICTABLE.

A5.2.8   Load and Store Word or Unsigned Byte - Immediate post-indexed

31	28	27	26	25	24	23	22	21	20	19	16	15	12	11	0
cond		0	1	0	0	U	B	0	L	Rn		Rd		offset_12	

This addressing mode uses the value of the base register Rn as the address for the memory access.

If the condition specified in the instruction matches the condition code status, the value of the immediate offset is added to or subtracted from the value of the base register Rn and written back to the base register Rn. The conditions are defined in *The condition field* on page A3-7.

Syntax

```
[<Rn>], #+/-<offset_12>
```

where:

<Rn>   Specifies the register containing the base address.

<offset_12>   Specifies the immediate offset used with the value of Rn to form the address.

Architecture version

All

Operation

```
address = Rn
if ConditionPassed(cond) then
    if U == 1 then
        Rn = Rn + offset_12
    else /* U == 0 */
        Rn = Rn - offset_12
```

Usage

This addressing mode is used for pointer access to arrays with automatic update of the pointer value.

Notes

Post-indexed addressing modes

LDRBT, LDRT, STRBT, and STRT only support post-indexed addressing modes. They use a minor modification of the above bit pattern, where bit[21] (the W bit) is 1, not 0 as shown.

Offset of zero   The syntax [<Rn>] is treated as an abbreviation for [<Rn>] , #0 for instructions that only support post-indexed addressing modes (LDRBT, LDRT, STRBT, STRT), but not for other instructions.

The B bit   This bit distinguishes between an unsigned byte (B==1) and a word (B==0) access.

The L bit   This bit distinguishes between a Load (L==1) and a Store (L==0) instruction.

Use of R15   Specifying R15 as register Rn has UNPREDICTABLE results.

ARM DDI 0100F-02

A5.2.9   Load and Store Word or Unsigned Byte - Register post-indexed

31 28	27	26	25	24	23	22	21	20 19 16	15 12	11 10 9 8 7 6 5 4	3 0
cond	0	1	1	0	U	B	0	L Rn	Rd	0 0 0 0 0 0 0 0	Rm

This addressing mode uses the value of the base register Rn as the address for the memory access.

If the condition specified in the instruction matches the condition code status, the value of the index register Rm is added to or subtracted from the value of the base register Rn and written back to the base register Rn. The conditions are defined in *The condition field* on page A3-7.

Syntax

```
[<Rn>], +/-<Rm>
```

where:

<Rn>   Specifies the register containing the base address.

<Rm>   Specifies the register containing the offset to add to or subtract from Rn.

Architecture version

All

Operation

```
address = Rn
if ConditionPassed(cond) then
    if U == 1 then
        Rn = Rn + Rm
    else /* U == 0 */
        Rn = Rn - Rm
```

Notes

Encoding　　　　This addressing mode is encoded as an `LSL` scaled register offset, scaled by zero.

Post-indexed addressing modes

　　　　　　　　`LDRBT`, `LDRT`, `STRBT`, and `STRT` only support post-indexed addressing modes. They use a minor modification of the above bit pattern, where bit[21] (the W bit) is 1, not 0 as shown.

The B bit　　　　This bit distinguishes between an unsigned byte (B==1) and a word (B==0) access.

The L bit　　　　This bit distinguishes between a Load (L==1) and a Store (L==0) instruction.

Use of R15　　　　Specifying R15 as register Rn or Rm has UNPREDICTABLE results.

Operand restriction　There are no operand restrictions in ARM architecture version 6 and above. In earlier versions of the architecture, if the same register is specified for Rn and Rm, the result is UNPREDICTABLE.

A5.2.10 Load and Store Word or Unsigned Byte - Scaled register post-indexed
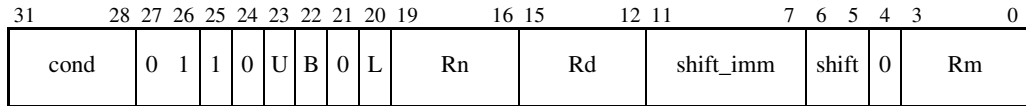
31 28	27 26	25	24	23	22	21	20	19 16	15 12	11 7	6 5	4	3 0
cond	0 1	1	0	U	B	0	L	Rn	Rd	shift_imm	shift	0	Rm

This addressing mode uses the value of the base register Rn as the address for the memory access.

If the condition specified in the instruction matches the condition code status, the shifted or rotated value of index register Rm is added to or subtracted from the value of the base register Rn and written back to the base register Rn. The conditions are defined in *The condition field* on page A3-7.

Syntax

One of:

```
[<Rn>], +/-<Rm>, LSL #<shift_imm>
[<Rn>], +/-<Rm>, LSR #<shift_imm>
[<Rn>], +/-<Rm>, ASR #<shift_imm>
[<Rn>], +/-<Rm>, ROR #<shift_imm>
[<Rn>], +/-<Rm>, RRX
```

where:

<Rn>	Specifies the register containing the base address.
<Rm>	Specifies the register containing the offset to add to or subtract from Rn.
LSL	Specifies a logical shift left.
LSR	Specifies a logical shift right.
ASR	Specifies an arithmetic shift right.
ROR	Specifies a rotate right.
RRX	Specifies a rotate right with extend.
<shift_imm>	Specifies the shift or rotation.

	LSL	0 to 31, encoded directly in the shift_imm field.
	LSR	1 to 32. A shift amount of 32 is encoded as shift_imm == 0. Other shift amounts are encoded directly.
	ASR	1 to 32. A shift amount of 32 is encoded as shift_imm == 0. Other shift amounts are encoded directly.
	ROR	1 to 31, encoded directly in the shift_imm field. (The shift_imm == 0 encoding is used to specify the RRX option.)

Architecture version

All

Operation

```
address = Rn
case shift of
    0b00 /* LSL */
        index = Rm Logical_Shift_Left shift_imm
    0b01 /* LSR */
        if shift_imm == 0 then /* LSR #32 */
            index = 0
        else
            index = Rm Logical_Shift_Right shift_imm
    0b10 /* ASR */
        if shift_imm == 0 then /* ASR #32 */
            if Rm[31] == 1 then
                index = 0xFFFFFFFF
            else
                index = 0
        else
            index = Rm Arithmetic_Shift_Right shift_imm
    0b11 /* ROR or RRX */
        if shift_imm == 0 then /* RRX */
            index = (C Flag Logical_Shift_Left 31) OR
                    (Rm Logical_Shift_Right 1)
        else /* ROR */
            index = Rm Rotate_Right shift_imm
endcase
if ConditionPassed(cond) then
    if U == 1 then
        Rn = Rn + index
    else /* U == 0 */
        Rn = Rn - index
```

Notes

The W bit	LDRBT, LDRT, STRBT, and STRT only support post-indexed addressing modes. They use a minor modification of the above bit pattern, where bit[21] (the W bit) is 1, not 0 as shown.
The B bit	This bit distinguishes between an unsigned byte (B == 1) and a word (B == 0) access.
The L bit	This bit distinguishes between a Load (L == 1) and a Store (L == 0) instruction.
Use of R15	Specifying R15 as register Rm or Rn has UNPREDICTABLE results.
Operand restriction	There are no operand restrictions in ARM architecture version 6 and above. In earlier versions of the architecture, if the same register is specified for Rn and Rm, the result is UNPREDICTABLE.

A5.3 Addressing Mode 3 - Miscellaneous Loads and Stores

There are six addressing modes used to calculate the address for load and store (signed or unsigned) halfword, load signed byte, or load and store doubleword instructions. The general instruction syntax is:

```
LDR|STR{<cond>}H|SH|SB|D  <Rd>, <addressing_mode>
```

where `<addressing_mode>` is one of the following six options:

1. `[<Rn>, #+/-<offset_8>]`

    See *Miscellaneous Loads and Stores - Immediate offset* on page A5-40.

2. `[<Rn>, +/-<Rm>]`

    See *Miscellaneous Loads and Stores - Register offset* on page A5-42.

3. `[<Rn>, #+/-<offset_8>]!`

    See *Miscellaneous Loads and Stores - Immediate pre-indexed* on page A5-44.

4. `[<Rn>, +/-<Rm>]!`

    See *Miscellaneous Loads and Stores - Register pre-indexed* on page A5-46.

5. `[<Rn>], #+/-<offset_8>`

    See *Miscellaneous Loads and Stores - Immediate post-indexed* on page A5-48.

6. `[<Rn>], +/-<Rm>`

    See *Miscellaneous Loads and Stores - Register post-indexed* on page A5-50.

A5.3.1 Encoding

The following diagrams show the encodings for this addressing mode:

Immediate offset/index

31 28	27 26 25	24	23	22	21	20	19 16	15 12	11 8	7	6	5	4	3 0
cond	0 0 0	P	U	1	W	L	Rn	Rd	immedH	1	S	H	1	ImmedL

Register offset/index

31 28	27 26 25	24	23	22	21	20	19 16	15 12	11 8	7	6	5	4	3 0
cond	0 0 0	P	U	0	W	L	Rn	Rd	SBZ	1	S	H	1	Rm

The P bit Has two meanings:

    P == 0   Indicates the use of *post-indexed addressing*. The base register value is used for the memory address, and the offset is then applied to the base register value and written back to the base register.

    P == 1   Indicates the use of *offset addressing* or *pre-indexed addressing* (the W bit determines which). The memory address is generated by applying the offset to the base register value.

The U bit Indicates whether the offset is added to the base (U == 1) or subtracted from the base (U == 0).

The W bit Has two meanings:

    P == 0   The W bit must be 0 or the instruction is UNPREDICTABLE.

    P == 1   W == 1 indicates that the memory address is written back to the base register (pre-indexed addressing), and W == 0 that the base register is unchanged (offset addressing).

The L, S and H bits

These bits combine to specify signed or unsigned loads or stores, and doubleword, halfword, or byte accesses:

    L=0, S=0, H=1   Store halfword.

    L=0, S=1, H=0   Load doubleword.

    L=0, S=1, H=1   Store doubleword.

    L=1, S=0, H=1   Load unsigned halfword.

    L=1, S=1, H=0   Load signed byte.

    L=1, S=1, H=1   Load signed halfword.

Signed bytes and halfwords can be stored with the same STRB and STRH instructions as are used for unsigned quantities, so no separate signed store instructions are provided.

Unsigned bytes

If S == 0 and H == 0, apparently indicating an unsigned byte, the instruction is not one that uses this addressing mode. Instead, it is a multiply instruction, a SWP or SWPB instruction, an LDREX or STREX instruction, or an unallocated instruction in the arithmetic or load/store instruction extension space (see *Extending the instruction set* on page A3-35).

Unsigned bytes are accessed by the LDRB, LDRBT, STRB and STRBT instructions, which use addressing mode 2 rather than addressing mode 3.

A5.3.2   Miscellaneous Loads and Stores - Immediate offset
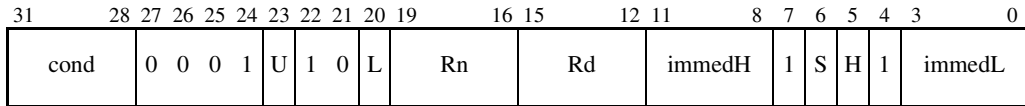
31 28	27 26 25 24	23	22 21	20	19 16	15 12	11 8	7	6	5	4	3 0
cond	0 0 0 1	U	1 0	L	Rn	Rd	immedH	1	S	H	1	immedL

This addressing mode calculates an address by adding or subtracting the value of an immediate offset to or from the value of the base register Rn.

Syntax

```
[<Rn>, #+/-<offset_8>]
```

where:

`<Rn>`	Specifies the register containing the base address.
`<offset_8>`	Specifies the immediate offset used with the value of Rn to form the address. The offset is encoded in immedH (top 4 bits) and immedL (bottom 4 bits).

Architecture version

Version 4 and above

Operation

```
offset_8 = (immedH << 4) OR immedL
if U == 1 then
    address = Rn + offset_8
else /* U == 0 */
    address = Rn - offset_8
```

Usage

This addressing mode is used for accessing structure (record) fields, and accessing parameters and locals variable in a stack frame. With an offset of zero, the address produced is the unaltered value of the base register Rn.

Notes

Zero offset	The syntax `[<Rn>]` is treated as an abbreviation for `[<Rn>,#0]`.
The L bit	This bit distinguishes between a Load (L==1) and a Store (L==0) instruction.
The S bit	This bit distinguishes between a signed (S==1) and an unsigned (S==0) halfword access.
The H bit	This bit distinguishes between a halfword (H==1) and a signed byte (H==0) access.

Use of R15   If R15 is specified as register Rn, the value used is the address of the instruction plus 8.

Unsigned bytes

If S == 0 and H == 0, apparently indicating an unsigned byte, the instruction is not one that uses this addressing mode. Instead, it is a multiply instruction, a SWP or SWPB instruction, or an unallocated instruction in the arithmetic or load/store instruction extension space (see *Extending the instruction set* on page A3-35).

Unsigned bytes are accessed by the LDRB, LDRBT, STRB and STRBT instructions, which use addressing mode 2 rather than addressing mode 3.

Signed stores   If S == 1 and L == 0, apparently indicating a signed store instruction, the instruction is an unallocated instruction in the load/store extension space (see *Extending the instruction set* on page A3-35).

Signed bytes and halfwords can be stored with the same STRB and STRH instructions as are used for unsigned quantities, so no separate signed store instructions are provided.

A5.3.3 Miscellaneous Loads and Stores - Register offset
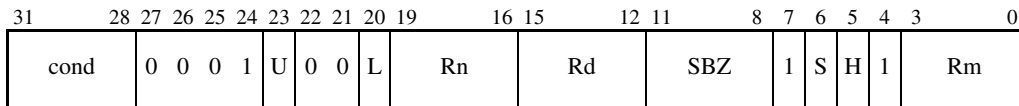
31	28	27 26 25 24	23	22 21	20	19	16	15	12	11	8	7	6	5	4	3	0
cond		0 0 0 1	U	0 0	L	Rn		Rd		SBZ		1	S	H	1	Rm	

This addressing mode calculates an address by adding or subtracting the value of the index register Rm to or from the value of the base register Rn.

Syntax

```
[<Rn>, +/-<Rm>]
```

where:

<Rn>   Specifies the register containing the base address.

<Rm>   Specifies the register containing the offset to add to or subtract from Rn.

Architecture version

Version 4 and above

Operation

```
if U == 1 then
    address = Rn + Rm
else /* U == 0 */
    address = Rn - Rm
```

Usage

This addressing mode is useful for pointer plus offset arithmetic and for accessing a single element of an array.

Notes

The L bit   Distinguishes between a Load (L==1) and a Store (L==0) instruction.

The S bit   Distinguishes between a signed (S==1) and an unsigned (S==0) halfword access.

The H bit   This bit distinguishes between a halfword (H==1) and a signed byte (H==0) access.

Unsigned bytes If S == 0 and H == 0, apparently indicating an unsigned byte, the instruction is not one that uses this addressing mode. Instead, it is a multiply instruction, a SWP or SWPB instruction, or an unallocated instruction in the arithmetic or load/store instruction extension space (see *Extending the instruction set* on page A3-35).

Unsigned bytes are accessed by the LDRB, LDRBT, STRB and STRBT instructions, which use addressing mode 2 rather than addressing mode 3.

Signed stores   If S == 1 and L == 0, apparently indicating a signed store instruction, the instruction is an unallocated instruction in the load/store extension space (see *Extending the instruction set* on page A3-35).

Signed bytes and halfwords can be stored with the same STRB and STRH instructions as are used for unsigned quantities, so no separate signed store instructions are provided.

Use of R15   If R15 is specified as register Rn, the value used is the address of the instruction plus 8. Specifying R15 as register Rm has UNPREDICTABLE results.

A5.3.4   Miscellaneous Loads and Stores - Immediate pre-indexed

31 28	27 26 25 24	23	22 21	20	19 16	15 12	11 8	7	6	5	4	3 0
cond	0 0 0 1	U	1 1	L	Rn	Rd	immedH	1	S	H	1	ImmedL

This addressing mode calculates an address by adding or subtracting the value of an immediate offset to or from the value of the base register `Rn`.

If the condition specified in the instruction matches the condition code status, the calculated address is written back to the base register Rn. The conditions are defined in *The condition field* on page A3-7.

Syntax

```
[<Rn>, #+/-<offset_8>]!
```

where:

`<Rn>`	Specifies the register containing the base address.
`<offset_8>`	Specifies the immediate offset used with the value of Rn to form the address. The offset is encoded in immedH (top 4 bits) and immedL (bottom 4 bits).
`!`	Sets the W bit, causing base register update.

Architecture version

Version 4 and above

Operation

```
offset_8 = (immedH << 4) OR immedL
if U == 1 then
    address = Rn + offset_8
else /* U == 0 */
    address = Rn - offset_8
if ConditionPassed(cond) then
    Rn = address
```

Usage

This addressing mode gives pointer access to arrays, with automatic update of the pointer value.

Notes

Offset of zero   The syntax [<Rn>] must not be treated as an abbreviation for [<Rn>,#0]!.

The L bit   This bit distinguishes between a Load (L==1) and a Store (L==0) instruction.

The S bit   This bit distinguishes between a signed (S==1) and an unsigned (S==0) halfword access.

The H bit   This bit distinguishes between a halfword (H==1) and a signed byte (H==0) access.

Unsigned bytes   If S == 0 and H == 0, apparently indicating an unsigned byte, the instruction is not one that uses this addressing mode. Instead, it is a multiply instruction, a SWP or SWPB instruction, or an unallocated instruction in the arithmetic or load/store instruction extension space (see *Extending the instruction set* on page A3-35).

Unsigned bytes are accessed by the LDRB, LDRBT, STRB and STRBT instructions, which use addressing mode 2 rather than addressing mode 3.

Signed stores   If S == 1 and L == 0, apparently indicating a signed store instruction, the instruction is an unallocated instruction in the load/store extension space (see *Extending the instruction set* on page A3-35).

Signed bytes and halfwords can be stored with the same STRB and STRH instructions as are used for unsigned quantities, so no separate signed store instructions are provided.

Use of R15   Specifying R15 as register Rn has UNPREDICTABLE results.

A5.3.5 Miscellaneous Loads and Stores - Register pre-indexed

31	28	27	26	25	24	23	22	21	20	19	16	15	12	11	8	7	6	5	4	3	0
cond		0	0	0	1	U	0	1	L	Rn		Rd		SBZ		1	S	H	1	Rm	

This addressing mode calculates an address by adding or subtracting the value of the index register Rm to or from the value of the base register Rn.

If the condition specified in the instruction matches the condition code status, the calculated address is written back to the base register Rn. The conditions are defined in *The condition field* on page A3-7.

Syntax

```
[<Rn>, +/-<Rm>]!
```

where:

`<Rn>`	Specifies the register containing the base address.
`<Rm>`	Specifies the register containing the offset to add to or subtract from Rn.
`!`	Sets the W bit, causing base register update.

Architecture version

Version 4 and above

Operation

```
if U == 1 then
    address = Rn + Rm
else /* U == 0 */
    address = Rn - Rm
if ConditionPassed(cond) then
    Rn = address
```

Notes

The L bit   This bit distinguishes between a Load (L==1) and a Store (L==0) instruction.

The S bit   This bit distinguishes between a signed (S==1) and an unsigned (S==0) halfword access.

The H bit   This bit distinguishes between a halfword (H==1) and a signed byte (H==0) access.

Unsigned bytes   If S == 0 and H == 0, apparently indicating an unsigned byte, the instruction is not one that uses this addressing mode. Instead, it is a multiply instruction, a SWP or SWPB instruction, or an unallocated instruction in the arithmetic or load/store instruction extension space (see *Extending the instruction set* on page A3-35).

Unsigned bytes are accessed by the LDRB, LDRBT, STRB and STRBT instructions, which use addressing mode 2 rather than addressing mode 3.

Signed stores   If S == 1 and L == 0, apparently indicating a signed store instruction, the instruction is an unallocated instruction in the load/store extension space (see *Extending the instruction set* on page A3-35).

Signed bytes and halfwords can be stored with the same STRB and STRH instructions as are used for unsigned quantities, so no separate signed store instructions are provided.

Use of R15   Specifying R15 as register Rm or Rn has UNPREDICTABLE results.

Operand restriction   There are no operand restrictions in ARM architecture version 6 and above. In earlier versions of the architecture, if the same register is specified for Rn and Rm, the result is UNPREDICTABLE.

A5.3.6   Miscellaneous Loads and Stores - Immediate post-indexed

31	28	27	26	25	24	23	22	21	20	19	16	15	12	11	8	7	6	5	4	3	0
cond		0	0	0	0	U	1	0	L	Rn		Rd		immedH		1	S	H	1	ImmedL	

This addressing mode uses the value of the base register Rn as the address for the memory access.

If the condition specified in the instruction matches the condition code status, the value of the immediate offset is added to or subtracted from the value of the base register Rn and written back to the base register Rn. The conditions are defined in *The condition field* on page A3-7.

Syntax

```
[<Rn>], #+/-<offset_8>
```

where:

<Rn>   Specifies the register containing the base address.

<offset_8>   Specifies the immediate offset used with the value of Rn to form the address. The offset is encoded in immedH (top 4 bits) and immedL (bottom 4 bits).

Architecture version

Version 4 and above

Operation

```
address = Rn
offset_8 = (immedH << 4) OR immedL
if ConditionPassed(cond) then
    if U == 1 then
        Rn = Rn + offset_8
    else /* U == 0 */
        Rn = Rn - offset_8
```

Usage

This addressing mode gives pointer access to arrays, with automatic update of the pointer value.

Notes

Offset of zero   The syntax `[<Rn>]` must not be treated as an abbreviation for `[<Rn>],#0`.

The L bit   This bit distinguishes between a Load (L==1) and a Store (L==0) instruction.

The S bit   This bit distinguishes between a signed (S==1) and an unsigned (S==0) halfword access.

The H bit   This bit distinguishes between a halfword (H==1) and a signed byte (H==0) access.

Unsigned bytes   If S == 0 and H == 0, apparently indicating an unsigned byte, the instruction is not one that uses this addressing mode. Instead, it is a multiply instruction, a `SWP` or `SWPB` instruction, or an unallocated instruction in the arithmetic or load/store instruction extension space (see *Extending the instruction set* on page A3-35).

Unsigned bytes are accessed by the `LDRB`, `LDRBT`, `STRB` and `STRBT` instructions, which use addressing mode 2 rather than addressing mode 3.

Signed stores   If S == 1 and L == 0, apparently indicating a signed store instruction, the instruction is an unallocated instruction in the load/store extension space (see *Extending the instruction set* on page A3-35).

Signed bytes and halfwords can be stored with the same `STRB` and `STRH` instructions as are used for unsigned quantities, so no separate signed store instructions are provided.

Use of R15   Specifying R15 as register Rn has UNPREDICTABLE results.

    Copyright © 1996–2003 ARM Limited. All rights reserved.
Confidential - Draft

A5.3.7 Miscellaneous Loads and Stores - Register post-indexed

31	28	27	26	25	24	23	22	21	20	19	16	15	12	11	8	7	6	5	4	3	0
cond		0	0	0	0	U	0	0	L	Rn		Rd		SBZ		1	S	H	1	Rm	

This addressing mode uses the value of the base register Rn as the address for the memory access.

If the condition specified in the instruction matches the condition code status, the value of the index register Rm is added to or subtracted from the value of the base register Rn and written back to the base register Rn. The conditions are defined in *The condition field* on page A3-7.

Syntax

```
[<Rn>], +/-<Rm>
```

where:

<Rn>   Specifies the register containing the base address.

<Rm>   Specifies the register containing the offset to add to or subtract from Rn.

Architecture version

Version 4 and above

Operation

```
address = Rn
if ConditionPassed(cond) then
    if U == 1 then
        Rn = Rn + Rm
    else /* U == 0 */
        Rn = Rn - Rm
```

Notes

The L bit   This bit distinguishes between a Load (L==1) and a Store (L==0) instruction.

The S bit   This bit distinguishes between a signed (S==1) and an unsigned (S==0) halfword access.

The H bit   This bit distinguishes between a halfword (H==1) and a signed byte (H==0) access.

Unsigned bytes   If S == 0 and H == 0, apparently indicating an unsigned byte, the instruction is not one that uses this addressing mode. Instead, it is a multiply instruction, a SWP or SWPB instruction, or an unallocated instruction in the arithmetic or load/store instruction extension space (see *Extending the instruction set* on page A3-35).

Unsigned bytes are accessed by the LDRB, LDRBT, STRB and STRBT instructions, which use addressing mode 2 rather than addressing mode 3.

Signed stores   If S == 1 and L == 0, apparently indicating a signed store instruction, the instruction is an unallocated instruction in the load/store extension space (see *Extending the instruction set* on page A3-35).

Signed bytes and halfwords can be stored with the same STRB and STRH instructions as are used for unsigned quantities, so no separate signed store instructions are provided.

Use of R15   Specifying R15 as register Rm or Rn has UNPREDICTABLE results.

Operand restriction   There are no operand restrictions in ARM architecture version 6 and above. In earlier versions of the architecture, if the same register is specified for Rn and Rm, the result is UNPREDICTABLE.

A5.4 Addressing Mode 4 - Load and Store Multiple

Load Multiple instructions load a subset (possibly all) of the general-purpose registers from memory. Store Multiple instructions store a subset (possibly all) of the general-purpose registers to memory.

Load and Store Multiple addressing modes produce a sequential range of addresses. The lowest-numbered register is stored at the lowest memory address and the highest-numbered register at the highest memory address.

The general instruction syntax is:

```
LDM|STM{<cond>}<addressing_mode> <Rn>{!}, <registers>{^}
```

where `<addressing_mode>` is one of the following four addressing modes:

1. `IA` (Increment After)

   See *Load and Store Multiple - Increment after* on page A5-54.

2. `IB` (Increment Before)

   See *Load and Store Multiple - Increment before* on page A5-55.

3. `DA` (Decrement After)

   See *Load and Store Multiple - Decrement after* on page A5-56.

4. `DB` (Decrement Before)

   See *Load and Store Multiple - Decrement before* on page A5-57.

There are also alternative mnemonics for these addressing modes, useful when `LDM` and `STM` are being used to access a stack, see *Load and Store Multiple addressing modes (alternative names)* on page A5-58.

A5.4.1 Encoding

The following diagram shows the encoding for this addressing mode:

31 28	27 26 25	24	23	22	21	20	19 16	15 0
cond	1 0 0	P	U	S	W	L	Rn	register list
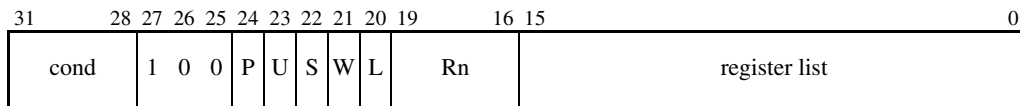
The P bit   Has two meanings:

    P==1   indicates that the word addressed by Rn is included in the range of memory locations accessed, lying at the top (U==0) or bottom (U==1) of that range.

    P==0   indicates that the word addressed by Rn is excluded from the range of memory locations accessed, and lies one word beyond the top of the range (U==0) or one word below the bottom of the range (U==1).

The U bit   Indicates that the transfer is made upwards (U==1) or downwards (U==0) from the base register.

The S bit   For LDMs that load the PC, the S bit indicates that the CPSR is loaded from the SPSR. For LDMs that do not load the PC and all STMs, the S bit indicates that when the processor is in a privileged mode, the User mode banked registers are transferred instead of the registers of the current mode.

LDM with the S bit set is UNPREDICTABLE in User or System mode.

The W bit   Indicates that the base register is updated after the transfer. The base register is incremented (U==1) or decremented (U==0) by four times the number of registers in the register list.

The L bit   Distinguishes between Load (L==1) and Store (L==0) instructions.

Register list   The register_list field of the instruction has one bit for each general-purpose register: bit[0] for register zero through to bit[15] for register 15 (the PC). If no bits are set, the result is UNPREDICTABLE.

The instruction syntax specifies the registers to load or store in <registers>, which is a comma-separated list of registers, surrounded by { and }.

A5.4.2   Load and Store Multiple - Increment after
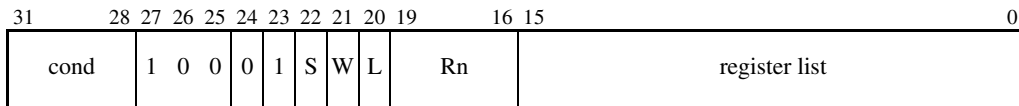
31	28	27	26	25	24	23	22	21	20	19	16	15	0

```
  cond      1  0  0 |0 |1 |S |W |L |     Rn     |              register list
```

This addressing mode is for Load and Store Multiple instructions, and forms a range of addresses.

The first address formed is the `<start_address>`, and is the value of the base register Rn. Subsequent addresses are formed by incrementing the previous address by four. One address is produced for each register that is specified in `<registers>`.

The last address produced is the `<end_address>`. Its value is four less than the sum of the value of the base register and four times the number of registers specified in `<registers>`.

If the condition specified in the instruction matches the condition code status and the W bit is set, Rn is incremented by four times the number of registers in `<registers>`. The conditions are defined in *The condition field* on page A3-7.

Syntax

```
IA
```

See also the alternative syntax described in *Load and Store Multiple addressing modes (alternative names)* on page A5-58.

Architecture version

All

Operation

```
start_address = Rn
end_address = Rn + (Number_Of_Set_Bits_In(register_list) * 4) - 4
if ConditionPassed(cond) and W == 1 then
    Rn = Rn + (Number_Of_Set_Bits_In(register_list) * 4)
```

Notes

The L bit   This bit distinguishes between a Load Multiple and a Store Multiple.

The S bit   For LDMs that load the PC, the S bit indicates that the CPSR is loaded from the SPSR. For LDMs that do not load the PC and all STMs, the S bit indicates that when the processor is in a privileged mode, the User mode banked registers are transferred instead of the registers of the current mode.

LDM with the S bit set is UNPREDICTABLE in User or System mode.

A5.4.3   Load and Store Multiple - Increment before
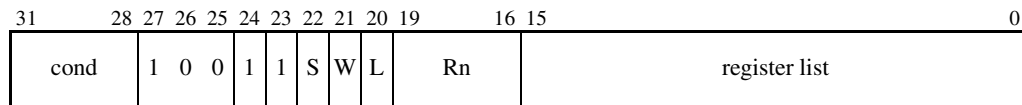
31	28	27	26	25	24	23	22	21	20	19	16	15	0
cond		1	0	0	1	1	S	W	L	Rn		register list	

This addressing mode is for Load and Store Multiple instructions, and forms a range of addresses.

The first address formed is the `<start_address>`, and is the value of the base register Rn plus four. Subsequent addresses are formed by incrementing the previous address by four. One address is produced for each register that is specified in `<registers>`.

The last address produced is the `<end_address>`. Its value is the sum of the value of the base register and four times the number of registers specified in `<registers>`.

If the condition specified in the instruction matches the condition code status and the W bit is set, Rn is incremented by four times the number of registers in `<registers>`. The conditions are defined in *The condition field* on page A3-7.

Syntax

```
IB
```

See also the alternative syntax described in *Load and Store Multiple addressing modes (alternative names)* on page A5-58.

Architecture version

All

Operation

```
start_address = Rn + 4
end_address = Rn + (Number_Of_Set_Bits_In(register_list) * 4)
if ConditionPassed(cond) and W == 1 then
    Rn = Rn + (Number_Of_Set_Bits_In(register_list) * 4)
```

Notes

The L bit This bit distinguishes between a Load Multiple and a Store Multiple.

The S bit For LDMs that load the PC, the S bit indicates that the CPSR is loaded from the SPSR. For LDMs that do not load the PC and all STMs, the S bit indicates that when the processor is in a privileged mode, the User mode banked registers are transferred instead of the registers of the current mode.

LDM with the S bit set is UNPREDICTABLE in User or System mode.

A5.4.4 Load and Store Multiple - Decrement after
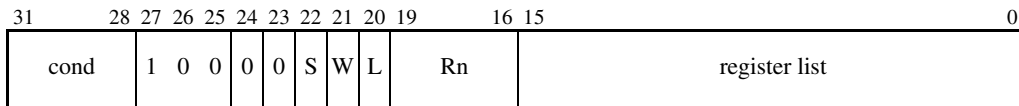
31	28	27 26 25	24 23	22	21	20	19	16	15	0
cond		1 0 0	0 0	S	W	L	Rn		register list	

This addressing mode is for Load and Store Multiple instructions, and forms a range of addresses.

The first address formed is the `<start_address>`, and is the value of the base register minus four times the number of registers specified in `<registers>`, plus 4. Subsequent addresses are formed by incrementing the previous address by four. One address is produced for each register that is specified in `<registers>`.

The last address produced is the `<end_address>`. Its value is the value of the base register Rn.

If the condition specified in the instruction matches the condition code status and the W bit is set, Rn is decremented by four times the number of registers in `<registers>`. The conditions are defined in *The condition field* on page A3-7.

Syntax

```
DA
```

See also the alternative syntax described in *Load and Store Multiple addressing modes (alternative names)* on page A5-58.

Architecture version

All

Operation

```
start_address = Rn - (Number_Of_Set_Bits_In(register_list) * 4) + 4
end_address = Rn
if ConditionPassed(cond) and W == 1 then
    Rn = Rn - (Number_Of_Set_Bits_In(register_list) * 4)
```

Notes

The L bit   This bit distinguishes between a Load Multiple and a Store Multiple.

The S bit   For LDMs that load the PC, the S bit indicates that the CPSR is loaded from the SPSR. For LDMs that do not load the PC and all STMs, the S bit indicates that when the processor is in a privileged mode, the User mode banked registers are transferred instead of the registers of the current mode.

LDM with the S bit set is UNPREDICTABLE in User or System mode.

A5.4.5 Load and Store Multiple - Decrement before
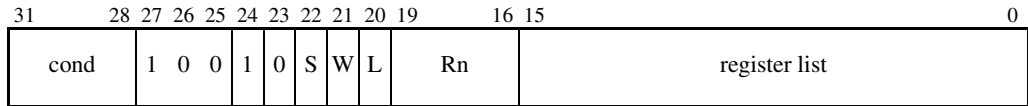
31 28	27	26	25	24	23	22	21	20	19 16	15 0
cond	1	0	0	1	0	S	W	L	Rn	register list

This addressing mode is for Load and Store multiple instructions, and forms a range of addresses.

The first address formed is the `<start_address>`, and is the value of the base register minus four times the number of registers specified in `<registers>`. Subsequent addresses are formed by incrementing the previous address by four. One address is produced for each register that is specified in `<registers>`.

The last address produced is the `<end_address>`. Its value is the value of the base register Rn minus four.

If the condition specified in the instruction matches the condition code status and the W bit is set, Rn is decremented by four times the number of registers in `<registers>`. The conditions are defined in *The condition field* on page A3-7.

Syntax

DB

See also the alternative syntax described in *Load and Store Multiple addressing modes (alternative names)* on page A5-58.

Architecture version

All

Operation

```
start_address = Rn - (Number_Of_Set_Bits_In(register_list) * 4)
end_address = Rn - 4
if ConditionPassed(cond) and W == 1 then
    Rn = Rn - (Number_Of_Set_Bits_In(register_list) * 4)
```

Notes

The L bit   This bit distinguishes between a Load Multiple and a Store Multiple.

The S bit   For LDMs that load the PC, the S bit indicates that the CPSR is loaded from the SPSR. For LDMs that do not load the PC and all STMs, the S bit indicates that when the processor is in a privileged mode, the User mode banked registers are transferred instead of the registers of the current mode.

LDM with the S bit set is UNPREDICTABLE in User or System mode.

A5.4.6 Load and Store Multiple addressing modes (alternative names)

The four addressing mode names given in *Addressing Mode 4 - Load and Store Multiple* on page A5-52 (IA, IB, DA, DB) are most useful when a load and Store Multiple instruction is being used for block data transfer, as it is likely that the Load Multiple and Store Multiple have the same addressing mode, so that the data is stored in the same way that it was loaded.

However, if Load Multiple and Store Multiple are being used to access a stack, the data is not loaded with the same addressing mode that was used to store the data, because the load (pop) and store (push) operations must adjust the stack in opposite directions.

Stack operations

Load Multiple and Store Multiple addressing modes can be specified with an alternative syntax, which is more applicable to stack operations:

Full stacks   Have stack pointers that point to the last used (full) location.

Empty stacks   Have stack pointers that point to the first unused (empty) location.

Descending stacks   Grow towards decreasing memory addresses (towards the bottom of memory).

Ascending stacks   Grow towards increasing memory addresses (towards the top of memory).

Two attributes allow four types of stack to be defined:
- Full Descending, with the syntax FD
- Empty Descending, with the syntax ED
- Full Ascending, with the syntax FA
- Empty Ascending, with the syntax EA.

———— **Note** ————

When defining stacks on which coprocessor data is to be placed (or might be placed in the future), programmers are advised to use the FD or EA stack types. This is because coprocessor data can be pushed to these types of stack with a single STC instruction and popped from them with a single LDC instruction. Multi-instruction sequences are required for coprocessor access to FA or ED stacks.

Table A5-1 on page A5-59 and Table A5-2 on page A5-59 show the relationship between the four types of stack, the four types of addressing mode shown above, and the L, U, and P bits in the instruction format.

Table A5-1 shows the relationship for LDM instructions.

Table A5-1 LDM addressing modes

Non-stack addressing mode	Stack addressing mode	L bit	P bit	U bit
LDMDA (Decrement After)	LDMFA (Full Ascending)	1	0	0
LDMIA (Increment After)	LDMFD (Full Descending)	1	0	1
LDMDB (Decrement Before)	LDMEA (Empty Ascending)	1	1	0
LDMIB (Increment Before)	LDMED (Empty Descending)	1	1	1

Table A5-2 shows the relationship for STM instructions.

Table A5-2 STM addressing modes

Non-stack addressing mode	Stack addressing mode	L bit	P bit	U bit
STMDA (Decrement After)	STMED (Empty Descending)	0	0	0
STMIA (Increment After)	STMEA (Empty Ascending)	0	0	1
STMDB (Decrement Before)	STMFD (Full Descending)	0	1	0
STMIB (Increment Before)	STMFA (Full Ascending)	0	1	1

A5.5   Addressing Mode 5 - Load and Store Coprocessor

There are four addressing modes which are used to calculate the address of a Load or Store Coprocessor instruction. The general instruction syntax is:

```
<opcode>{<cond>}{L}   <coproc>,<CRd>,<addressing_mode>
```

where `<addressing_mode>` is one of the following four options:

1.   `[<Rn>,#+/-<offset_8>*4]`

      See *Load and Store Coprocessor - Immediate offset* on page A5-62.

2.   `[<Rn>,#+/-<offset_8>*4]!`

      See *Load and Store Coprocessor - Immediate pre-indexed* on page A5-64.

3.   `[<Rn>],#+/-<offset_8>*4`

      See *Load and Store Coprocessor - Immediate post-indexed* on page A5-66.

4.   `[<Rn>],<option>`

      See *Load and Store Coprocessor - Unindexed* on page A5-68.

A5.5.1   Encoding

The following diagram shows the encoding for this addressing mode:

31 28	27 26 25	24	23	22	21	20	19 16	15 12	11 8	7 0
cond	1 1 0	P	U	N	W	L	Rn	CRd	cp#	offset_8
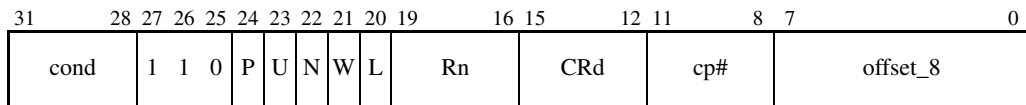
The P bit   Has two meanings:

        P == 0   Indicates the use of *post-indexed addressing* or *unindexed addressing* (the W bit determines which). The base register value is used for the memory address.

        P == 1   Indicates the use of *offset addressing* or *pre-indexed addressing* (the W bit determines which). The memory address is generated by applying the offset to the base register value.

The U bit   Has two meanings:

        U == 1   Indicates that the offset is added to the base.

        U == 0   Indicates that the offset is subtracted from the base

The N bit   The meaning of this bit is coprocessor-dependent. Its recommended use is to distinguish between different-sized values to be transferred.

The W bit   Has two meanings:

        W == 1   Indicates that the memory address is written back to the base register.

        W == 0   Indicates that the base register value is unchanged.

Also:

- If P == 0, this distinguishes unindexed addressing (W == 0) from post-indexed addressing (W == 1). For unindexed addressing, U must equal 1 or the result is either UNDEFINED or UNPREDICTABLE (see *Coprocessor instruction extension space* on page A3-41).

- If P == 1, this distinguishes offset addressing (W == 0) from pre-indexed addressing (W == 1).

The L bit   Distinguishes between Load (L == 1) and Store (L == 0) instructions.

A5.5.2 Load and Store Coprocessor - Immediate offset

31	28	27	26	25	24	23	22	21	20	19	16	15	12	11	8	7	0
cond		1	1	0	1	U	N	0	L	Rn		CRd		cp_num		offset_8	

This addressing mode produces a sequence of consecutive addresses. The first address is calculated by adding or subtracting four times the value of an immediate offset to or from the value of the base register Rn. The subsequent addresses in the sequence are produced by incrementing the previous address by four until the coprocessor signals the end of the instruction. This allows a coprocessor to access data whose size is coprocessor-defined.

The coprocessor must not request a transfer of more than 16 words.

Syntax

```
[<Rn>, #+/-<offset_8>*4]
```

where:

<Rn>   Specifies the register containing the base address.

<offset_8>   Specifies the immediate offset that is multiplied by 4, then added to or subtracted
                from the value of Rn to form the address.

Architecture version

Version 2 and above

Operation

```
if ConditionPassed(cond) then
    if U == 1 then
        address = Rn + offset_8 * 4
    else /* U == 0 */
        address = Rn - offset_8 * 4
    start_address = address
    while (NotFinished(coprocessor[cp_num]))
        address = address + 4
    end_address = address
```

ARM DDI 0100F-02

Notes

The N bit   Is coprocessor-dependent.

The L bit   Distinguishes between Load (L==1) and Store (L==0) instructions.

Use of R15   If R15 is specified as register Rn, the value used is the address of the instruction plus 8.

A5.5.3   Load and Store Coprocessor - Immediate pre-indexed
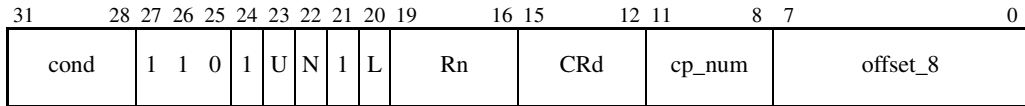
31	28 27 26 25 24 23 22 21 20 19	16 15	12 11	8 7	0

cond	1 1 0 1	U	N	1	L	Rn	CRd	cp_num	offset_8

This addressing mode produces a sequence of consecutive addresses. The first address is calculated by adding or subtracting four times the value of an immediate offset to or from the value of the base register Rn. If the condition specified in the instruction matches the condition code status, the first address is written back to the base register Rn. The subsequent addresses in the sequence are produced by incrementing the previous address by four until the coprocessor signals the end of the instruction. This allows a coprocessor to access data whose size is coprocessor-defined.

The coprocessor must not request a transfer of more than 16 words.

Syntax

```
[<Rn>, #+/-<offset_8>*4]!
```

where:

`<Rn>`	Specifies the register containing the base address.
`<offset_8>`	Specifies the immediate offset that is multiplied by 4, then added to or subtracted from the value of Rn to form the address.
`!`	Sets the W bit, causing base register update.

Architecture version

Version 2 and above

Operation

```
if ConditionPassed(cond) then
    if U == 1 then
        Rn = Rn + offset_8 * 4
    else /* U == 0 */
        Rn = Rn - offset_8 * 4
    start_address = Rn
    address = start_address
    while (NotFinished(coprocessor[cp_num]))
        address = address + 4
    end_address = address
```

Notes

The N bit   Is coprocessor-dependent.

The L bit   Distinguishes between Load (L==1) and Store (L==0) instructions.

Use of R15   Specifying R15 as register Rn has UNPREDICTABLE results.

A5.5.4 Load and Store Coprocessor - Immediate post-indexed
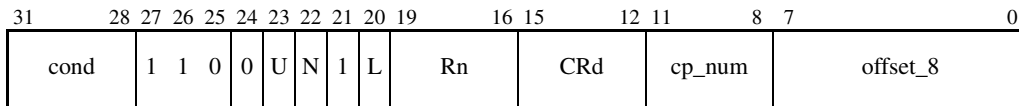
31 28	27 26 25 24 23	22	21	20	19 16	15 12	11 8	7 0
cond	1 1 0 0	U	N	1	L	Rn	CRd	cp_num	offset_8

(bit field: 31–28 cond; 27 26 25 24 = 1 1 0 0; 23 U; 22 N; 21 = 1; 20 L; 19–16 Rn; 15–12 CRd; 11–8 cp_num; 7–0 offset_8)

This addressing mode produces a sequence of consecutive addresses. The first address is the value of the base register Rn. The subsequent addresses in the sequence are produced by incrementing the previous address by four until the coprocessor signals the end of the instruction. This allows a coprocessor to access data whose size is coprocessor-defined.

If the condition specified in the instruction matches the condition code status, the base register Rn is updated by adding or subtracting four times the value of an immediate offset to or from the value of the base register Rn.

The coprocessor must not request a transfer of more than 16 words.

Syntax

```
[<Rn>], #+/-<offset_8>*4
```

where:

`<Rn>`	Specifies the register containing the base address.
`<offset_8>`	Specifies the immediate offset that is multiplied by 4, then added to or subtracted from the value of Rn to form the address.

Architecture version

Version 2 and above

Operation

```
if ConditionPassed(cond) then
    start_address = Rn
    if U == 1 then
        Rn = Rn + offset_8 * 4
    else /* U == 0 */
        Rn = Rn - offset_8 * 4
    address = start_address
    while (NotFinished(coprocessor[cp_num]))
        address = address + 4
    end_address = address
```

Notes

The N bit   Is coprocessor-dependent.

The L bit   Distinguishes between Load (L==1) and Store (L==0) instructions.

Use of R15   Specifying R15 as register Rn has UNPREDICTABLE results.

A5.5.5 Load and Store Coprocessor - Unindexed
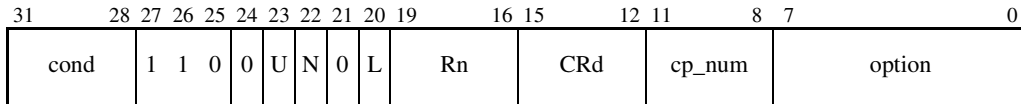
31	28	27	26	25	24	23	22	21	20	19				16	15		12	11		8	7		0
cond		1	1	0	0	U	N	0	L	Rn					CRd			cp_num			option		

This addressing mode produces a sequence of consecutive addresses. The first address is the value of the base register Rn. The subsequent addresses in the sequence are produced by incrementing the previous address by four until the coprocessor signals the end of the instruction. This allows a coprocessor to access data whose size is coprocessor-defined.

The base register Rn is not updated. Bits[7:0] of the instruction are therefore not used by the ARM, either for the address calculation or to calculate a new value for the base register, and so can be used to specify additional instruction options to the coprocessor.

The coprocessor must not request a transfer of more than 16 words.

Syntax

```
[<Rn>], <option>
```

where:

<Rn>   Specifies the register containing the base address.

<option>   Specifies additional instruction options to the coprocessor. The <option> is specified in
            the instruction syntax as an integer in the range 0-255, surrounded by { and }.

Architecture version

Version 2 and above

Operation

```
if ConditionPassed(cond) then
    start_address = Rn
    address = start_address
    while (NotFinished(coprocessor[cp_num]))
        address = address + 4
    end_address = address
```

Notes

The N bit	Is coprocessor-dependent.
The L bit	Distinguishes between Load (L==1) and Store (L==0) instructions.
Use of R15	If R15 is specified as register Rn, the value used is the address of the instruction plus 8.
The U bit	If bit[23] (the Up/down bit) is not set, the result is either UNDEFINED or UNPREDICTABLE (see *Coprocessor instruction extension space* on page A3-41).
Option bits	Are unused by the ARM in this addressing mode, and therefore can be used to request additional instruction options in a coprocessor-dependent fashion.

Chapter A6
The Thumb Instruction Set

This chapter introduces the Thumb instruction set and describes how Thumb uses the ARM programmer's model. It contains the following sections:

A6.1 About the Thumb instruction set

The Thumb instruction set is a re-encoded subset of the ARM instruction set. Thumb is designed to increase the performance of ARM implementations that use a 16-bit or narrower memory data bus and to allow better code density than provided by the ARM instruction set. T variants of the ARM architecture incorporate both a full 32-bit ARM instruction set and the 16-bit Thumb instruction set. Every Thumb instruction is encoded in 16 bits.

Thumb does not alter the underlying programmer's model of the ARM architecture. It merely presents restricted access to it. All Thumb data-processing instructions operate on full 32-bit values, and full 32-bit addresses are produced by both data-access instructions and instruction fetches.

When the processor is executing Thumb instructions, eight general-purpose integer registers are available, R0 to R7, which are the same physical registers as R0 to R7 when executing ARM instructions. Some Thumb instructions also access the Program Counter (ARM register 15), the Link Register (ARM register 14) and the Stack Pointer (ARM register 13). Further instructions allow limited access to ARM registers 8 to 15, which are known as the *high registers*.

When R15 is read, bit[0] is zero and bits[31:1] contain the PC. When R15 is written, bit[0] is IGNORED and bits[31:1] are written to the PC. Depending on how it is used, the value of the PC is either the address of the instruction plus 4 or is UNPREDICTABLE.

Thumb execution is flagged by the T bit (bit[5]) in the CPSR:

T == 0   32-bit instructions are fetched (and the PC is incremented by four) and are executed as ARM instructions.

T == 1   16-bit instructions are fetched (and the PC is incremented by two) and are executed as Thumb instructions.

In architecture v6, the Thumb instruction set provides limited access to the CPSR with the CPS instruction. There is no direct access to the SPSRs. Earlier versions provided no direct access to the CPSR. (In the ARM instruction set, the MSR and MRS instructions, and CPS in architecture v6, do this.)

A6.1.1 Entering Thumb state

Thumb execution is normally entered by executing an ARM BX instruction (Branch and Exchange). This instruction branches to the address held in a general-purpose register, and if bit[0] of that register is 1, Thumb execution begins at the branch target address. If bit[0] of the target register is 0, ARM execution continues from the branch target address. On architecture versions 5 and above, BLX instructions and LDR/LDM instructions that load the PC can be used similarly.

Thumb execution can also be initiated by setting the T bit in the SPSR and executing an ARM instruction which restores the CPSR from the SPSR (a data-processing instruction with the S bit set and the PC as the destination, or a Load Multiple with Restore CPSR instruction). This allows an operating system to automatically restart a process independent of whether that process is executing Thumb code or ARM code.

The result is UNPREDICTABLE if the T bit is altered directly by writing the CPSR.

A6.1.2   Exceptions

Exceptions generated during Thumb execution switch to ARM execution before executing the exception
handler (whose first instruction is at the hardware vector). The state of the T bit is preserved in the SPSR,
and the LR of the exception mode is set so that the normal return instruction performs correctly, regardless
of whether the exception occurred during ARM or Thumb execution. Table A6-1 lists the values of the
exception mode LR for exceptions generated during Thumb execution.

Table A6-1 Exception return instructions

Exception	Exception link register value	Return instruction
Reset	UNPREDICTABLE value	-
Undefined	Address of undefined instruction + 2	MOVS PC, R14
SWI	Address of SWI instruction + 2	MOVS PC, R14
Prefetch Abort	Address of aborted instruction fetch + 4	SUBS PC, R14, #4
Data Abort	Address of the instruction that generated the abort + 8	SUBS PC, R14, #8
IRQ	Address of the next instruction to be executed + 4	SUBS PC, R14, #4
FIQ	Address of the next instruction to be executed + 4	SUBS PC, R14, #4

——— **Note** ———

For each exception, the return instruction indicated by Table 6-1 is the same as the return instruction
required if the exception occurred during ARM execution, for the primary or only method of return from
that instruction listed in *Exceptions* on page A2-15. However, the following two types of exception have a
secondary return method, for which different return instructions are needed depending on whether the
exception occurred during ARM or Thumb execution:

- For the Data Abort exception, the primary method of return causes execution to resume at the aborted
  instruction, which causes it to be re-executed. As described in *Data Abort (data access memory
  abort)* on page A2-20, it is also possible to return to the next instruction after the aborted instruction,
  using a SUBS PC,R14,#4 instruction. If this type of return is required for a Data Abort caused by
  a Thumb instruction, use SUBS PC,R14,#6 for the return instruction.

- For the Undefined Instruction exception, the primary method of return causes execution to resume at
  the next instruction after the undefined instruction. As described in *Undefined Instruction exception*
  on page A2-18, it is also possible to return to the undefined instruction itself, using the instruction
  SUBS PC,R14,#4. If this type of return is required for a Thumb undefined instruction, use SUBS
  PC,R14,#2 for the return instruction. However, the main use of this type of return is for some types
  of coprocessor instruction, and as the Thumb instruction set does not contain any coprocessor
  instructions, you are unlikely to need this secondary method of return for Thumb instructions.

When these secondary methods of return are used, the exception handler code must test the SPSR T bit to
determine which of the two return instructions to use.

A6.2 Instruction set encoding

Figure A6-1 shows the Thumb instruction set encoding. An entry in square brackets, for example [1], indicates a note on the following page.

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Shift by immediate	0	0	0	opcode [1]		immediate					Rm			Rd		
Add/subtract register	0	0	0	1	1	0	opc		Rm		Rn			Rd		
Add/subtract immediate	0	0	0	1	1	1	opc		immediate		Rn			Rd		
Add/subtract/compare/move immediate	0	0	1	opcode		Rd / Rn			immediate							
Data-processing register	0	1	0	0	0	0	opcode				Rm / Rs			Rd / Rn		
Special data processing	0	1	0	0	0	1	opcode [1]		H1	H2	Rm			Rd / Rn		
Branch/exchange instruction set [3]	0	1	0	0	0	1	1	1	L	H2	Rm			SBZ		
Load from literal pool	0	1	0	0	1	Rd			PC-relative offset							
Load/store register offset	0	1	0	1	opcode		Rm		Rn		Rd					
Load/store word/byte immediate offset	0	1	1	B	L	offset					Rn			Rd		
Load/store halfword immediate offset	1	0	0	0	L	offset					Rn			Rd		
Load/store to/from stack	1	0	0	1	L	Rd			SP-relative offset							
Add to SP or PC	1	0	1	0	SP	Rd			immediate							
Miscellaneous: See Figure 6-2	1	0	1	1	x	x	x	x	x	x	x	x	x	x	x	x
Load/store multiple	1	1	0	0	L	Rn			register list							
Conditional branch	1	1	0	1	cond [2]				offset							
Undefined instruction	1	1	0	1	1	1	1	0	x	x	x	x	x	x	x	x
Software interrupt	1	1	0	1	1	1	1	1	immediate							
Unconditional branch	1	1	1	0	0	offset										
BLX suffix [4]	1	1	1	0	1	offset										0
Undefined instruction	1	1	1	0	1	x	x	x	x	x	x	x	x	x	x	1
BL/BLX prefix	1	1	1	1	0	offset										
BL suffix	1	1	1	1	1	offset										

Figure A6-1 Thumb instruction set overview

1. The `opc` field is not allowed to be 11 in this line. Other lines deal with the case that the `opc` field is 11.

2. The `cond` field is not allowed to be 1110 or 1111 in this line. Other lines deal with the cases where the `cond` field is 1110 or 1111.

3. The form with `L==1` is UNPREDICTABLE prior to ARM architecture version 5.

4. This is an undefined instruction prior to ARM architecture version 5.

A6.2.1 Miscellaneous instructions

Figure A6-2 lists miscellaneous Thumb instructions. An entry in square brackets, for example [1], indicates a note below the figure.

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Adjust stack pointer	1	0	1	1	0	0	0	0	opc	immediate						
Sign/zero extend [2]	1	0	1	1	0	0	1	0	opc		Rn			Rd		
Push/pop register list	1	0	1	1	L	1	0	R	register list							
Set Endianness [2]	1	0	1	1	0	1	1	0	0	1	0	1	E	SBZ		
Change Processor State [2]	1	0	1	1	0	1	1	0	0	1	1	imod	0	A	I	F
Reverse bytes [2]	1	0	1	1	1	0	1	0	opc		Rn			Rd		
Software breakpoint [1]	1	0	1	1	1	1	1	0	immediate							

Figure A6-2 Miscellaneous Thumb instructions

1. This is an undefined instruction prior to ARM architecture version 5.

2. These are undefined instructions prior to ARM architecture version 6.

──── **Note** ────

Any instruction with bits[15:12] = 1011, and which is not shown in Figure A6-2, is an undefined instruction.

A6.3 Branch instructions

Thumb supports six types of branch instruction:

- a conditional branch to allow forward and backward branches of up to 256 bytes

- an unconditional branch that allows a forward or backward branch of up to 2KB

- a Branch with Link (subroutine call) is supported with a pair of instructions that allow forward and backward branches of up to 4MB

- a Branch with Link and Exchange uses a pair of instructions, similar to Branch with Link, but additionally switches to ARM code execution.

- a Branch and Exchange instruction branches to an address in a register and optionally switches to ARM code execution

- a second form of Branch with Link and Exchange instruction performs a subroutine call to an address in a register and optionally switches to ARM code execution

The encoding for these instructions is given below.

A6.3.1 Conditional branch

```
B<cond>   <target_address>
```

15	14	13	12	11　　　　　　8	7　　　　　　　　　　　　　　0
1	1	0	1	cond	8_bit_signed_offset
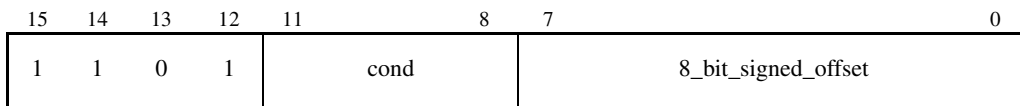
A6.3.2 Unconditional branch

```
B   <target_address>
BL  <target_address>              ; Produces two 16-bit instructions
BLX <target_address>              ; Produces two 16-bit instructions
```

15	14	13	12	11	10　　　　　　　　　　　　　　0
1	1	1	H		offset_11
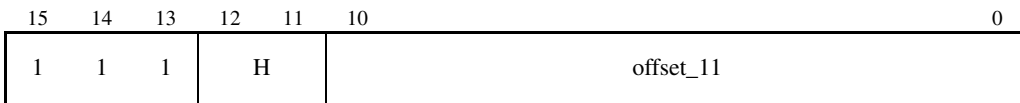
A6.3.3 Branch with exchange

```
BX    <Rm>
BLX   <Rm>
```

15	14	13	12	11	10	9	8	7	6	5　　　3	2　　　0
0	1	0	0	0	1	1	1	L	H2	Rm	SBZ
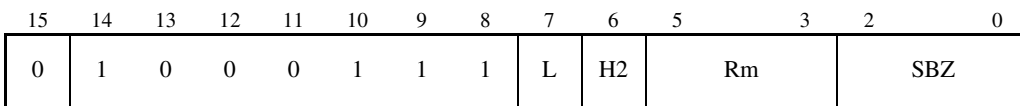
A6.3.4   Examples

```
     B    label        ; unconditionally branch to label

     BCC  label        ; branch to label if carry flag is clear

     BEQ  label        ; branch to label if zero flag is set

     BL   func         ; subroutine call to function
func
...                    ; Include body of function here
...
     MOV  PC, LR        ; R15=R14, return to instruction after the BL

     BX   R12          ; branch to address in R12; begin ARM execution if
                       ; bit 0 of R12 is zero; otherwise continue executing
                       ; Thumb code
```

A6.3.5   List of branch instructions

The following instructions follow the formats shown above.

B   Conditional Branch. See *B (1)* on page A7-19.

B   Unconditional Branch. See *B (2)* on page A7-21.

BL   Branch with Link. See *BL, BLX (1)* on page A7-26.

BX   Branch and Exchange instruction set. See *BX* on page A7-32.

BLX   Branch with Link and Exchange instruction set. See *BL, BLX (1)* on page A7-26 and *BLX (2)* on page A7-30.

A6.4 Data-processing instructions

Thumb data-processing instructions are a subset of the ARM data-processing instructions. They are divided into two sets. The first set can only operate on the low registers, r0-r7. The second set can operate on the high registers, r8-r15, or on a mixture of low and high registers.

A6.4.1 Low register data-processing instructions

The low register data processing instructions are shown in Table A6-2. Some of these instructions also appear in the high register data processing instruction list. When operating on low registers, all instructions in this table, except CPY, set the condition codes.

Table A6-2 Low register data-processing instructions

Mnemonic	Operation	Action
ADC Rd, Rm	Add with Carry	Rd := Rd + Rm + Carry flag
ADD Rd, Rn, Rm	Add	Rd := Rn + Rm
ADD Rd, Rn, #0 to 7	Add	Rd := Rn + 3-bit immediate
ADD Rd, #0 to 255	Add	Rd := Rd + 8-bit immediate
AND Rd, Rm	Logical AND	Rd := Rd AND Rm
ASR Rd, Rm, #1 to 32	Arithmetic Shift Right	Rd := Rm ASR 5-bit immediate
ASR Rd, Rs	Arithmetic Shift Right	Rd := Rd ASR Rs
BIC Rd, Rm	Bit Clear	Rd := Rd AND NOT Rm
CMN Rn, Rm	Compare Negated	Update flags after Rn + Rm
CMP Rn, #0 to 255	Compare	Update flags after Rn - 8-bit immediate
CMP Rn, Rm	Compare	Update flags after Rn - Rm
CPY Rd, Rn	Copy	Rd := Rn
EOR Rd, Rm	Logical Exclusive OR	Rd := Rd EOR Rm
LSL Rd, Rm, #0 to 31	Logical Shift Left	Rd := Rm LSL 5-bit immediate
LSL Rd, Rs	Logical Shift Left	Rd := Rd LSL Rs
LSR Rd, Rm, #1 to 32	Logical Shift Right	Rd := Rm LSR 5-bit immediate
LSR Rd, Rs	Logical Shift Right	Rd := Rd LSR Rs
MOV Rd, #0 to 255	Move	Rd := 8-bit immediate

Table A6-2 Low register data-processing instructions (continued)

Mnemonic	Operation	Action
`MOV Rd, Rn`	Move	Rd := Rn
`MUL Rd, Rm`	Multiply	Rd := Rm x Rd
`MVN Rd, Rm`	Move Not	Rd := NOT Rm
`NEG Rd, Rm`	Negate	Rd := 0 - Rm
`ORR Rd, Rm`	Logical (inclusive) OR	Rd := Rd OR Rm
`ROR Rd, Rs`	Rotate Right	Rd := Rd ROR Rs
`SBC Rd, Rm`	Subtract with Carry	Rd := Rd - Rm - NOT(Carry Flag)
`SUB Rd, Rn, Rm`	Subtract	Rd := Rn - Rm
`SUB Rd, Rn, #0 to 7`	Subtract	Rd := Rn - 3-bit immediate
`SUB Rd, #0 to 255`	Subtract	Rd := Rd - 8-bit immediate
`TST Rn, Rm`	Test	Update flags after Rn AND Rm

For example:

```
ADD    R0, R4, R7       ; R0 = R4 + R7
SUB    R6, R1, R2       ; R6 = R1 - R2
ADD    R0, #255         ; R0 = R0 + 255
ADD    R1, R4, #4       ; R1 = R4 + 4
NEG    R3, R1           ; R3 = 0 - R1
AND    R2, R5           ; R2 = R2 AND R5
EOR    R1, R6           ; R1 = R1 EOR R6
CMP    R2, R3           ; update flags after R2 - R3
CMP    R7, #100         ; update flags after R7 - 100
MOV    R0, #200         ; R0 = 200
```

A6.4.2 High registers

There are eight types of data-processing instruction which operate on ARM registers 8 to 14 and the PC as shown in Table A6-3. Apart from `CMP`, instructions in this table do not change the condition code flags.

Table A6-3 High register data-processing instructions

Mnemonic	Operation	Action
`MOV Rd, Rn`	Move	Rd := Rn
`CPY Rd, Rn`	Copy	Rd := Rn
`ADD Rd, Rm`	Add	Rd := Rd + Rm
`CMP Rn, Rm`	Compare	Update flags after Rn - Rm
`ADD SP, #0 to 508`	Increment stack pointer	R13 = R13 + 4* (7-bit immediate)
`SUB SP, #0 to 508`	Decrement stack pointer	R13 = R13 - 4* (7-bit immediate)
`ADD Rd, SP, #0 to 1020`	Form Stack address	Rd = R13 + 4* (8-bit immediate)
`ADD Rd, PC, #0 to 1020`	Form PC address	Rd = PC + 4* (8-bit immediate)

For example:

```
MOV     R0, R12             ; R0 = R12
ADD     R10, R1             ; R10 = R10 + R1
MOV     PC, LR              ; PC = R14
CMP     R10, R11            ; update flags after R10 - R11
SUB     SP, #12             ; increase stack size by 12 bytes
ADD     SP, #16             ; decrease stack size by 16 bytes
ADD     R2, SP, #20         ; R2 = SP + 20
ADD     R0, PC, #500        ; R0 = PC + 500
```

A6.4.3   Formats

Data-processing instructions use the following eight instruction formats:

Format 1

```
<opcode1>   <Rd>, <Rn>, <Rm>
<opcode1> := ADD | SUB
```

15	14	13	12	11	10	9	8	6	5	3	2	0
0	0	0	1	1	0	op_1	Rm		Rn		Rd	
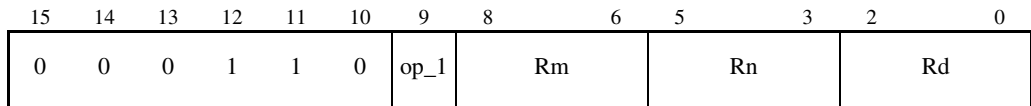
Format 2

```
<opcode2>   <Rd>, <Rn>, #<3_bit_immed>
<opcode2> := ADD | SUB
```

15	14	13	12	11	10	9	8	6	5	3	2	0
0	0	0	1	1	1	op_2	3_bit_immediate		Rn		Rd	
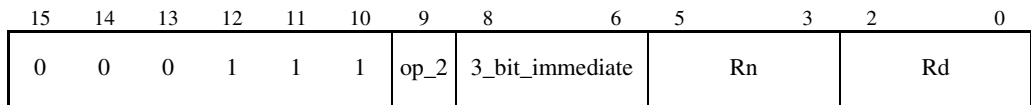
Format 3

```
<opcode3>   <Rd>|<Rn>, #<8_bit_immed>
<opcode3> := ADD | SUB | MOV | CMP
```

15	14	13	12	11	10	8	7	0
0	0	1	op_3		Rd|Rn		8_bit_immediate	

Format 4

```
<opcode4>   <Rd>, <Rm>, #<shift_imm>
<opcode4> := LSL | LSR | ASR
```

15	14	13	12	11	10	6	5	3	2	0
0	0	0	op_4		shift_immediate		Rm		Rd	
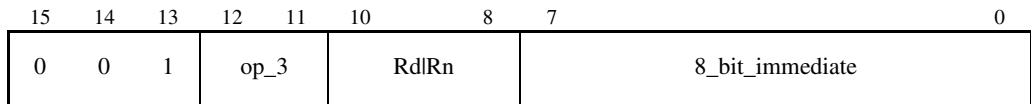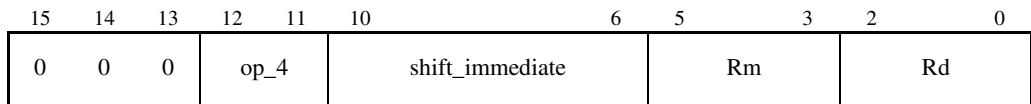
Format 5

```
<opcode5>  <Rd>|<Rn>, <Rm>|<Rs>
<opcode5> := MVN | CMP | CMN | TST | ADC | SBC | NEG | MUL |
             LSL | LSR | ASR | ROR | AND | EOR | ORR | BIC
```

15	14	13	12	11	10	9			6	5		3	2		0
0	1	0	0	0	0		op_5				Rm|Rs			Rd|Rn	
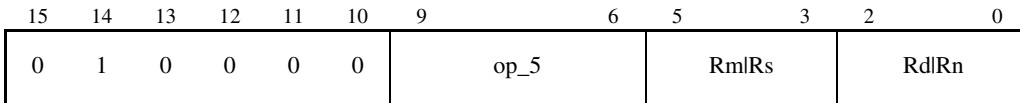
Format 6

```
ADD  <Rd>, <reg>, #<8_bit_immed>
<reg> := SP | PC
```

15	14	13	12	11	10		8	7							0
1	0	1	0	reg		Rd					8_bit_immediate				
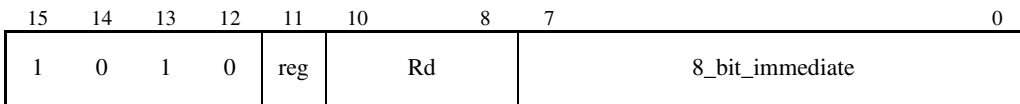
Format 7

```
<opcode6>  SP, SP, #<7_bit_immed>
<opcode6> := ADD | SUB
```

15	14	13	12	11	10	9	8	7	6						0
1	0	1	1	0	0	0	0	op_6			7_bit_immediate				

Format 8

```
<opcode7>  <Rd>|<Rn>, <Rm>
<opcode7> := MOV | ADD | CMP | CPY
```

15	14	13	12	11	10	9	8	7	6	5		3	2		0
0	1	0	0	0	1	opcode		H1	H2	Rm			Rd|Rn		
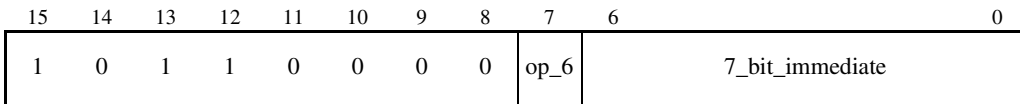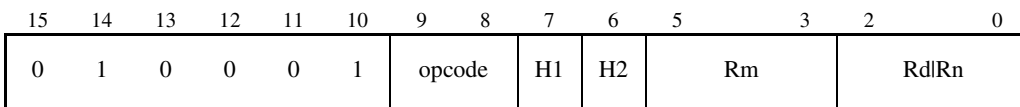
A6.4.4   List of data-processing instructions

The following instructions follow the formats shown above.

ADC	Add with Carry. See *ADC* on page A7-4.
ADD	Add (immediate). See *ADD (1)* on page A7-5.
ADD	Add (large immediate). See *ADD (2)* on page A7-6.
ADD	Add (register). See *ADD (3)* on page A7-7.
ADD	Add high registers. See *ADD (4)* on page A7-8.
ADD	Add (immediate to program counter). See *ADD (5)* on page A7-10.
ADD	Add (immediate to stack pointer). See *ADD (6)* on page A7-11.
ADD	Increment stack pointer. See *ADD (7)* on page A7-12.
AND	Logical AND. See *AND* on page A7-14.
ASR	Arithmetic Shift Right (immediate). See *ASR (1)* on page A7-15.
ASR	Arithmetic Shift Right (register). See *ASR (2)* on page A7-17.
BIC	Bit Clear. See *BIC* on page A7-23.
CMN	Compare Negative (register). See *CMN* on page A7-34.
CMP	Compare (immediate). See *CMP (1)* on page A7-35.
CMP	Compare (register). See *CMP (2)* on page A7-36.
CMP	Compare high registers. See *CMP (3)* on page A7-37.
CPY	Copy high or low registers. See *CPY* on page A7-41.
EOR	Exclusive OR. See *EOR* on page A7-43.
LSL	Logical Shift Left (immediate). See *LSL (1)* on page A7-63.
LSL	Logical Shift Left (register). See *LSL (2)* on page A7-65.
LSR	Logical Shift Right (immediate). See *LSR (1)* on page A7-67.
LSR	Logical Shift Right (register). See *LSR (2)* on page A7-69.
MOV	Move (immediate). See *MOV (1)* on page A7-71.
MOV	Move a low register to another low register. See *MOV (2)* on page A7-72.
MOV	Move high registers. See *MOV (3)* on page A7-74.
MUL	Multiply. See *MUL* on page A7-76.

MVN   Move NOT (register). See *MVN* on page A7-78.

NEG   Negate (register). See *NEG* on page A7-79.

ORR   Logical OR. See *ORR* on page A7-80.

ROR   Rotate Right (register). See *ROR* on page A7-90.

SBC   Subtract with Carry (register). See *SBC* on page A7-92.

SUB   Subtract (immediate). See *SUB (1)* on page A7-111.

SUB   Subtract (large immediate). See *SUB (2)* on page A7-112.

SUB   Subtract (register). See *SUB (3)* on page A7-113.

SUB   Decrement stack pointer. See *SUB (4)* on page A7-114.

TST   Test (register). See *TST* on page A7-118.

A6.5   Load and Store Register instructions

Thumb supports eight types of Load and Store Register instructions. Two basic addressing modes are available. These allow the load and store of words, halfwords and bytes, and also the load of signed halfwords and bytes:

• register plus register

• register plus 5-bit immediate (not available for signed halfword and signed byte loads).

If an immediate offset is used, it is scaled by 4 for word access and 2 for halfword accesses.

In addition, three special instructions allow:

• words to be loaded using the PC as a base with a 1KB (word-aligned) immediate offset

• words to be loaded and stored with the stack pointer (R13) as the base and a 1KB (word-aligned) immediate offset.

A6.5.1   Formats

Load and Store Register instructions have the following formats:

Format 1

```
<opcode1>  <Rd>, [<Rn>, #<5_bit_offset>]
<opcode1> := LDR|LDRH|LDRB|STR|STRH|STRB
```

15	11	10	6	5	3	2	0
opcode1		5_bit_offset		Rn		Rd	
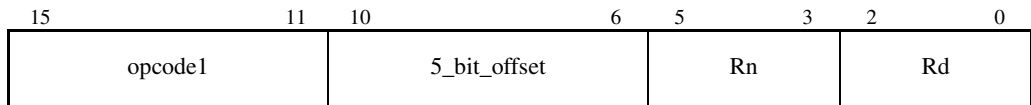
Format 2

```
<opcode2>  <Rd>, [<Rn>, <Rm>]
<opcode2> := LDR|LDRH|LDRSH|LDRB|LDRSB|STR|STRH|STRB
```

15	9	8	6	5	3	2	0
opcode2		Rm		Rn		Rd	
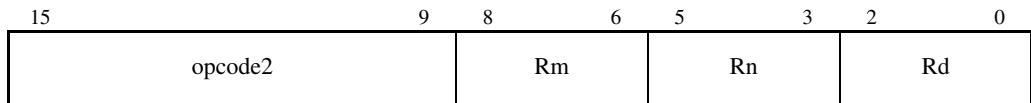
Format 3

```
LDR  <Rd>, [PC, #<8_bit_offset>]
```

15	14	13	12	11	10	8	7	0
0	1	0	0	1	Rd		8_bit_immediate	
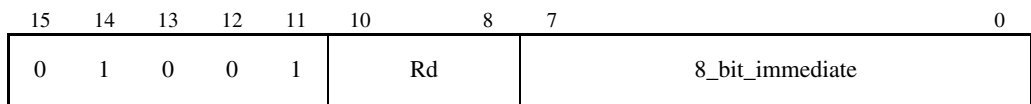
Format 4

```
<opcode3>  <Rd>, [SP, #<8_bit_offset>]
<opcode3>  := LDR | STR
```

15	14	13	12	11	10 8	7 0
1	0	0	1	L	Rd	8_bit_immediate
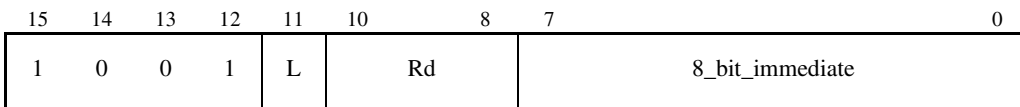
For example:

```
LDR    R4, [R2, #4]         ; Load word into R4 from address R2 + 4
LDR    R4, [R2, R1]         ; Load word into R4 from address R2 + R1
STR    R0, [R7, #0x7C]      ; Store word from R0 to address R7 + 124
STRB   R1, [R5, #31]        ; Store byte from R1 to address R5 + 31
STRH   R4, [R2, R3]         ; Store halfword from R4 to R2 + R3
LDRH   R3, [R6, R5]         ; Load word into R3 from R6 + R5
LDRB   R2, [R1, #5]         ; Load byte into R2 from R1 + 5
LDR    R6, [PC, #0x3FC]     ; Load R6 from PC + 0x3FC
LDR    R5, [SP, #64]        ; Load R5 from SP + 64
STR    R4, [SP, #0x260]     ; Load R5 from SP + 0x260
```

A6.5.2   List of Load and Store Register instructions

The following instructions follow the formats shown above.

LDR   Load Word (immediate offset). See *LDR (1)* on page A7-46.

LDR   Load Word (register offset). See *LDR (2)* on page A7-48.

LDR   Load Word (PC-relative). See *LDR (3)* on page A7-50.

LDR   Load Word (SP-relative). See *LDR (4)* on page A7-52.

LDRB   Load Unsigned Byte (immediate offset). See *LDRB (1)* on page A7-54.

LDRB   Load Unsigned Byte (register offset). See *LDRB (2)* on page A7-55.

LDRH   Load Unsigned Halfword (immediate offset). See *LDRH (1)* on page A7-56.

LDRH   Load Unsigned Halfword (register offset). See *LDRH (2)* on page A7-58.

LDRSB   Load Signed Byte (register offset). See *LDRSB* on page A7-60.

LDRSH   Load Signed Halfword (register offset). See *LDRSH* on page A7-61.

STR   Store Word (immediate offset). See *STR (1)* on page A7-98.

STR   Store Word (register offset). See *STR (2)* on page A7-100.

STR   Store Word (SP-relative). See *STR (3)* on page A7-102.

STRB   Store Byte (immediate offset). See *STRB (1)* on page A7-104.

STRB   Store Byte (register offset). See *STRB (2)* on page A7-106.

STRH   Store Halfword (immediate offset). See *STRH (1)* on page A7-107.

STRH   Store Halfword (register offset). See *STRH (2)* on page A7-109.

A6.6 Load and Store Multiple instructions

Thumb supports four types of Load and Store Multiple instructions:

- Two instructions, LDMIA and STMIA, are designed to support block copy. They have a fixed Increment After addressing mode from a base register.

- The other two instructions, PUSH and POP, also have a fixed addressing mode. They implement a full descending stack and the stack pointer (R13) is used as the base register.

All four instructions update the base register after transfer and all can transfer any or all of the lower 8 registers. PUSH can also stack the return address, and POP can load the PC.

A6.6.1 Formats

Load and Store Multiple instructions have the following formats:

Format 1

```
<opcode1>  <Rn>!, <registers>
<opcode1>  := LDMIA | STMIA
```

15	14	13	12	11	10		8	7		0
1	1	0	0	L		Rn			register_list	

Format 2

```
PUSH  {<registers>}
POP   {<registers>}
```

15	14	13	12	11	10	9	8	7		0
1	0	1	1	L	1	0	R		register_list	
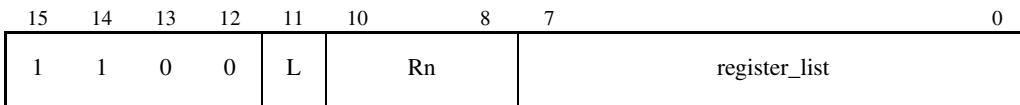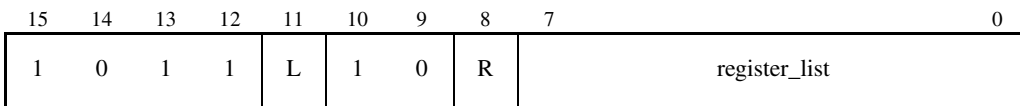
A6.6.2 Examples

```
 LDMIA   R7!, {R0-R3, R5}          ; Load R0 to R3-R5 from R7, add 20 to R7
    STMIA   R0!, {R3, R4, R5}         ; Store R3-R5 to R0: add 12 to R0

function
    PUSH    {R0-R7, LR}               ; push onto the stack (R13) R0-R7 and
                                      ; the return address
...                                   ; code of the function body
...
    POP     {R0-R7, PC}               ; restore R0-R7 from the stack
                                      ; and the program counter, and return
```

A6.6.3   List of Load and Store Multiple instructions

The following instructions follow the formats shown above.

LDMIA   Load Multiple. See *LDMIA* on page A7-44.

POP   Pop Multiple. See *POP* on page A7-81.

PUSH   Push Multiple. See *PUSH* on page A7-84.

STMIA   Store Multiple. See *STMIA* on page A7-96.

A6.7 Exception-generating instructions

The Thumb instruction set provides two types of instruction whose main purpose is to cause a processor exception to occur:

- The Software Interrupt (SWI) instruction is used to cause a SWI exception to occur (see *Software Interrupt exception* on page A2-19). This is the main mechanism in the Thumb instruction set by which User mode code can make calls to privileged Operating System code.

- The Breakpoint (BKPT) instruction is used for software breakpoints in T variants of ARM architecture versions 5 and above. Its default behavior is to cause a Prefetch Abort exception to occur (see *Prefetch Abort (instruction fetch memory abort)* on page A2-20). A debug monitor program that has previously been installed on the Prefetch Abort vector can handle this exception.

  If debug hardware is present in the system, it is allowed to override this default behavior. Details of whether and how this happens are IMPLEMENTATION DEFINED.

A6.7.1 Instruction encodings

```
SWI   <immed_8>
```

15	14	13	12	11	10	9	8	7	0
1	1	0	1	1	1	1	1	immed_8	

```
BKPT   <immed_8>
```

15	14	13	12	11	10	9	8	7	0
1	0	1	1	1	1	1	0	immed_8	
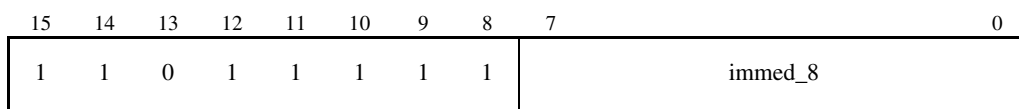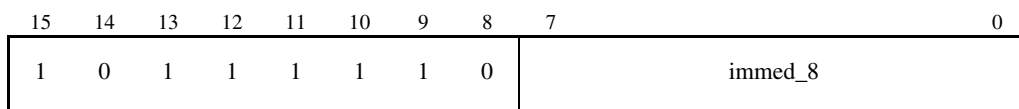
In both SWI and BKPT, the immed_8 field of the instruction is ignored by the ARM processor. The SWI or Prefetch Abort handler can optionally be written to load the instruction that caused the exception and extract these fields. This allows them to be used to communicate extra information about the Operating System call or breakpoint to the handler.

A6.7.2 List of exception-generating instructions

BKPT   Breakpoint. See *BKPT* on page A7-24.

SWI   Software Interrupt. See *SWI* on page A7-116.

A6.8 Undefined instruction space

The following instructions are UNDEFINED in the Thumb instruction set:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	0	0	0	1	x	x	x	x	x	x	x	x
1	0	1	1	0	x	1	1	x	x	x	x	x	x	x	x
1	0	1	1	1	0	0	x	x	x	x	x	x	x	x	x
1	0	1	1	1	0	1	0	1	0	x	x	x	x	x	x
1	0	1	1	1	0	1	1	x	x	x	x	x	x	x	x
1	0	1	1	1	1	1	1	x	x	x	x	x	x	x	x
1	1	0	1	1	1	1	0	x	x	x	x	x	x	x	x

In general, these instructions can be used to extend the Thumb instruction set in the future. However, it is intended that the following group of instructions will not be used in this manner:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	1	1	1	0	x	x	x	x	x	x	x	x

Use one of these instructions if you want to use an undefined instruction for software purposes, with minimal risk that future hardware will treat it as a defined instruction.

Chapter A7
Thumb Instructions

This chapter describes the syntax and usage of every Thumb instruction, in the section:

- *Alphabetical list of Thumb instructions* on page A7-2
- *Thumb instructions and architecture versions* on page A7-121.

A7.1 Alphabetical list of Thumb instructions

Every Thumb instruction is listed on the following pages. Each instruction description shows:

- the instruction encoding
- the instruction syntax
- the versions of the ARM architecture where the instruction is valid
- any exceptions that might apply
- a pseudo-code specification of how the instruction operates
- notes on usage and special cases
- the equivalent ARM instruction encoding.

A7.1.1 General notes

These notes explain the types of information and abbreviations used on the instruction pages.

Syntax abbreviations

The following abbreviations are used in the instruction pages:

immed_<n>   This is an <n>-bit immediate value. For example, an 8-bit immediate value is represented by:

        immed_8

signed_immed_<n>

        This is a signed immediate. For example, an 8-bit signed immediate is represented by:

        signed_immed_8

Architecture version

For the convenience of the reader, this section describes the version of the ARM architecture that the instruction is associated with, not the version of the Thumb instruction set. There have been three versions of the Thumb instruction set architecture to date:

THUMBv1   This is used in T variants of version 4 of the ARM instruction set architecture.

THUMBv2   This is used in T variants of version 5 of the ARM instruction set architecture.

THUMBv3   This is used in version 6 and above of the ARM instruction set architecture.

Instructions which are described as being in all T variants are therefore present in THUMBv1, THUMBv2, and THUMBv3. and those that are described as being in T variants of version 6 and above are in THUMBv3 only.

Equivalent ARM syntax and encoding

This section shows the syntax and encoding of an equivalent ARM instruction. When no precise equivalent is available, a close equivalent is shown and the reasons why it is not a precise equivalent are explained.

A common reason for the instruction not being a precise equivalent is that it reads the value of the PC. This produces the instruction's own address plus N, where N is 8 for ARM instructions and 4 for Thumb instructions. This difference can often be compensated for by adjusting an immediate constant in the equivalent ARM instruction.

In the equivalent instruction encodings, named fields and bits must be filled in with the corresponding fields and bits from the Thumb instruction, or in a few cases with values derived from the Thumb instruction as described in the text.

The ARM instruction fields are normally the same length as the corresponding Thumb instruction fields, with one important exception. Thumb register fields are normally 3 bits long, whereas ARM register fields are normally 4 bits long. In these cases, the Thumb register field must be extended with a high-order 0 when substituted into the ARM register field, so that the ARM instruction refers to the correct one of R0 to R7.

Information on usage

Usage information is only given for Thumb instructions where it differs significantly from ARM instruction usage. If no Usage section appears for a Thumb instruction, see the equivalent ARM instruction page in Chapter A4 *ARM Instructions* for usage information.

A7.1.2 ADC

15	14	13	12	11	10	9	8	7	6	5		3	2		0
0	1	0	0	0	0	0	1	0	1		Rm			Rd	

ADC (Add with Carry) adds two values and the Carry flag.

Use ADC to synthesize multi-word addition.

ADC updates the condition code flags, based on the result.

Syntax

```
ADC  <Rd>, <Rm>
```

where:

<Rd>   Holds the first value for the addition, and is the destination register for the operation.

<Rm>   Specifies the register that contains the second operand for the addition.

Architecture version

All T variants.

Exceptions

None.

Operation

```
Rd = Rd + Rm + C Flag
N Flag = Rd[31]
Z Flag = if Rd == 0 then 1 else 0
C Flag = CarryFrom(Rd + Rm + C Flag)
V Flag = OverflowFrom(Rd + Rm + C Flag)
```

Equivalent ARM syntax and encoding

```
ADCS  <Rd>, <Rd>, <Rm>
```

31 30 29 28	27 26 25 24 23 22 21 20	19 16	15 12	11 10 9 8 7 6 5 4	3 0
1 1 1 0	0 0 0 0 1 0 1 1	Rd	Rd	0 0 0 0 0 0 0 0	Rm

A7.1.3   ADD (1)

15	14	13	12	11	10	9	8		6	5		3	2		0
0	0	0	1	1	1	0		immed_3			Rn			Rd	

ADD (1) adds a small constant value to the value of a register and stores the result in a second register.

It updates the condition code flags, based on the result.

Syntax

```
ADD  <Rd>, <Rn>, #<immed_3>
```

where:

`<Rd>`	Is the destination register for the completed operation.
`<Rn>`	Specifies the register that contains the operand for the addition.
`<immed_3>`	Specifies a 3-bit immediate value that is added to the value of `<Rn>`.

Architecture version

All T variants.

Exceptions

None.

Operation

```
Rd = Rn + immed_3
N Flag = Rd[31]
Z Flag = if Rd == 0 then 1 else 0
C Flag = CarryFrom(Rn + immed_3)
V Flag = OverflowFrom(Rn + immed_3)
```

Equivalent ARM syntax and encoding

```
ADDS  <Rd>, <Rn>, #<immed_3>
```

31 30 29 28	27 26 25 24 23 22 21 20	19 16	15 12	11 10 9 8 7 6 5 4 3	2 0
1 1 1 0	0 0 1 0 1 0 0 1	Rn	Rd	0 0 0 0 0 0 0 0 0	immed_3

A7.1.4   ADD (2)

15	14	13	12	11	10		8	7		0
0	0	1	1	0		Rd			immed_8	
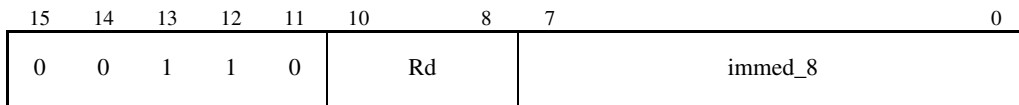
ADD (2) adds a large immediate value to the value of a register and stores the result back in the same register.

The condition code flags are updated, based on the result.

Syntax

```
ADD   <Rd>, #<immed_8>
```

where:

<Rd>   Holds the first operand for the addition, and is the destination register for the completed operation.

<immed_8>   Specifies an 8-bit immediate value that is added to the value of <Rd>.

Architecture version

All T variants.

Exceptions

None.

Operation

```
Rd = Rd + immed_8
N Flag = Rd[31]
Z Flag = if Rd == 0 then 1 else 0
C Flag = CarryFrom(Rd + immed_8)
V Flag = OverflowFrom(Rd + immed_8)
```

Equivalent ARM syntax and encoding

```
ADDS   <Rd>, <Rd>, #<immed_8>
```

31	30	29	28	27	26	25	24	23	22	21	20	19		16	15		12	11	10	9	8	7		0
1	1	1	0	0	0	1	0	1	0	0	1		Rd			Rd		0	0	0	0		immed_8	
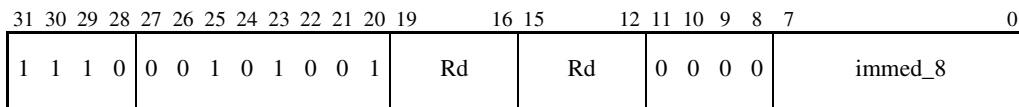
A7.1.5 ADD (3)

15	14	13	12	11	10	9	8 6	5 3	2 0
0	0	0	1	1	0	0	Rm	Rn	Rd

ADD (3) adds the value of one register to the value of a second register, and stores the result in a third register.

It updates the condition code flags, based on the result.

Syntax

```
ADD   <Rd>, <Rn>, <Rm>
```

where:

<Rd>   Is the destination register for the completed operation.

<Rn>   Specifies the register containing the first value for the addition.

<Rm>   Specifies the register containing the second value for the addition.

Architecture version

All T variants.

Exceptions

None.

Operation

```
Rd = Rn + Rm
N Flag = Rd[31]
Z Flag = if Rd == 0 then 1 else 0
C Flag = CarryFrom(Rn + Rm)
V Flag = OverflowFrom(Rn + Rm)
```

Equivalent ARM syntax and encoding

```
ADDS  <Rd>, <Rn>, <Rm>
```

31 30 29 28	27 26 25 24 23 22 21 20	19 16	15 12	11 10 9 8 7 6 5 4	3 0
1 1 1 0	0 0 0 0 1 0 0 1	Rn	Rd	0 0 0 0 0 0 0 0	Rm

A7.1.6   ADD (4)

15	14	13	12	11	10	9	8	7	6	5		3	2		0
0	1	0	0	0	1	0	0	H1	H2	Rm			Rd		

ADD (4) adds the values of two registers, one or both of which are high registers.

Unlike the low-register only ADD instruction (*ADD (3)* on page A7-7), this instruction does not change the flags.

Syntax

```
ADD   <Rd>, <Rm>
```

where:

<Rd>   Specifies the register containing the first value, and is also the destination register. It can be any of R0 to R15. The register number is encoded in the instruction in H1 (most significant bit) and Rd (remaining three bits).

<Rm>   Specifies the register containing the second value. It can be any of R0 to R15. Its number is encoded in the instruction in H2 (most significant bit) and Rm (remaining three bits).

Architecture version

All T variants.

Exceptions

None.

Operation

```
Rd = Rd + Rm
```

Notes

Operand restriction   If a low register is specified for <Rd> and Rm (H1==0 and H2==0), the result is UNPREDICTABLE.

                                   ARM DDI 0100F-02

Equivalent ARM syntax and encoding

A close equivalent is:

```
ADD   <Rd>, <Rd>, <Rm>
```

31 30 29 28	27 26 25 24 23 22 21 20	19 18	16 15 14	12 11 10 9 8 7 6 5 4 3 2	0
1 1 1 0	0 0 0 0 1 0 0 0	H1 Rd	H1 Rd	0 0 0 0 0 0 0 0 H2	Rm

There are slight differences when the instruction accesses the PC, because of the different definitions of the PC when executing ARM and Thumb code.

A7.1.7 ADD (5)

15	14	13	12	11	10	8	7	0
1	0	1	0	0	Rd		immed_8	
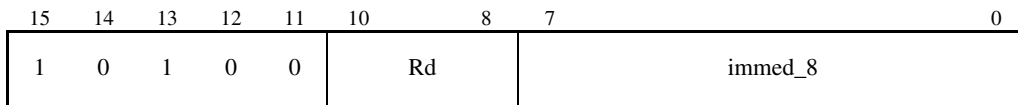
ADD (5) adds an immediate value to the PC and writes the resulting PC-relative address to a destination register. The immediate can be any multiple of 4 in the range 0 to 1020.

The condition codes are not affected.

Syntax

```
ADD   <Rd>, PC, #<immed_8> * 4
```

where:

<Rd>   Is the destination register for the completed operation.

PC   Indicates PC-relative addressing.

<immed_8> Specifies an 8-bit immediate value that is quadrupled and added to the value of the PC.

Architecture version

All T variants.

Exceptions

None.

Operation

```
Rd = (PC AND 0xFFFFFFFC) + (immed_8 << 2)
```

Equivalent ARM syntax and encoding

A close equivalent is:

```
ADD   <Rd>, PC, #<immed_8> * 4
```

31 30 29 28	27 26 25 24 23 22 21 20	19 18 17 16	15 12	11 10 9 8	7 0
1 1 1 0	0 0 1 0 1 0 0 0	1 1 1 1	Rd	1 1 1 1	immed_8
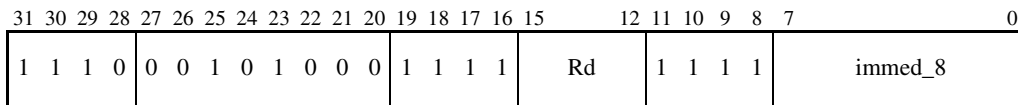
The definitions of the PC differ between ARM and Thumb code. This makes a difference between the precise results of the instructions.

A7.1.8   ADD (6)

15	14	13	12	11	10	8	7	0
1	0	1	0	1	Rd		immed_8	
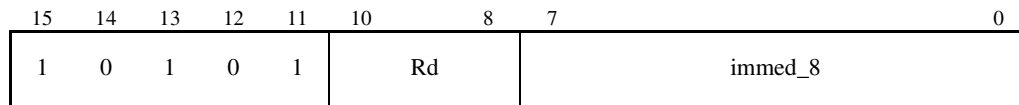
ADD (6) adds an immediate value to the SP and writes the resulting SP-relative address to a destination register. The immediate can be any multiple of 4 in the range 0 to 1020.

The condition codes are not affected.

Syntax

```
ADD   <Rd>, SP, #<immed_8> * 4
```

where:

<Rd>   Is the destination register for the completed operation.

SP   Indicates SP-relative addressing.

<immed_8>   Specifies an 8-bit immediate value that is quadrupled and added to the value of the SP.

Architecture version

All T variants.

Exceptions

None.

Operation

```
Rd = SP + (immed_8 << 2)
```

Equivalent ARM syntax and encoding

```
ADD   <Rd>, SP, #<immed_8> * 4
```

31 30 29 28	27 26 25 24 23 22 21 20	19 18 17 16	15 12	11 10 9 8	7 0
1 1 1 0	0 0 1 0 1 0 0 0	1 1 0 1	Rd	1 1 1 1	immed_8

A7.1.9   ADD (7)
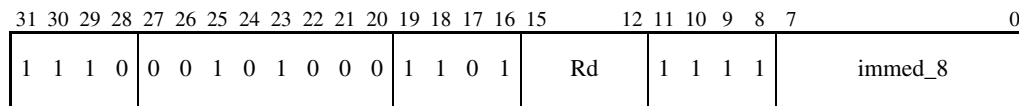
15	14	13	12	11	10	9	8	7	6		0
1	0	1	1	0	0	0	0	0		immed_7	
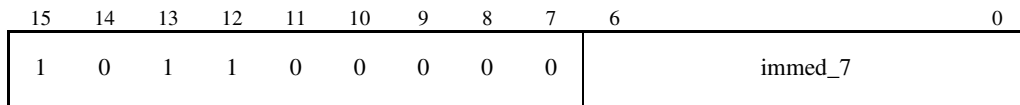
ADD (7) increments the SP by four times a 7-bit immediate (that is, by a multiple of 4 in the range 0 to 508).

The condition codes are not affected.

Syntax

```
ADD  SP, #<immed_7> * 4
```

where:

SP   Contains the first operand for the addition. SP is also the destination register for the operation.

<immed_7> Specifies the immediate value that is quadrupled and added to the value of the SP.

Architecture version

All T variants.

Exceptions

None.

Operation

```
SP = SP + (immed_7 << 2)
```

Usage

For the Full Descending stack which the Thumb instruction set is designed to use, incrementing the SP is used to discard data on the top of the stack.

Notes

Alternative syntax   This instruction can also be written as ADD SP, SP, #(<immed_7> * 4).

Equivalent ARM syntax and encoding

```
ADD  SP, SP, #<immed_7> * 4
```

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	0
1 1 1 0	0 0 1 0 1 0 0 0	1 1 0 1	1 1 0 1	1 1 1 1 0	immed_7

A7.1.10 AND

15	14	13	12	11	10	9	8	7	6	5		3	2		0
0	1	0	0	0	0	0	0	0	0	Rm			Rd		

AND (Logical AND) performs a bitwise AND of the values in two registers.

AND updates the condition code flags, based on the result.

Syntax

```
AND  <Rd>, <Rm>
```

where:

<Rd>   Specifies the register containing the first operand, and is also the destination register.

<Rm>   Specifies the register containing the second operand.

Architecture version

All T variants.

Exceptions

None.

Operation

```
Rd = Rd AND Rm
N Flag = Rd[31]
Z Flag = if Rd == 0 then 1 else 0
C Flag = unaffected
V Flag = unaffected
```

Equivalent ARM syntax and encoding

```
ANDS  <Rd>, <Rd>, <Rm>
```

31	30	29	28	27	26	25	24	23	22	21	20	19		16	15		12	11	10	9	8	7	6	5	4	3		0
1	1	1	0	0	0	0	0	0	0	0	1	Rd			Rd			0	0	0	0	0	0	0	0	Rm		

A7.1.11 ASR (1)

15	14	13	12	11	10	6	5	3	2	0
0	0	0	1	0	immed_5		Rm		Rd	
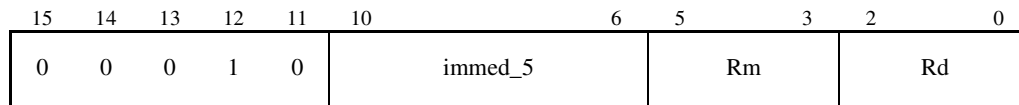
ASR (1) (Arithmetic Shift Right) provides the signed value of the contents of a register divided by a constant power of 2.

It updates the condition code flags, based on the result.

Syntax

```
ASR  <Rd>, <Rm>, #<immed_5>
```

where:

<Rd>	Is the destination register for the completed operation.
<Rm>	Specifies the register that contains the value to be shifted.
<immed_5>	Specifies the shift amount, in the range 1 to 32. Shifts by 1 to 31 are encoded directly in immed_5. A shift by 32 is encoded as immed_5 == 0.

Architecture version

All T variants.

Exceptions

None.

Operation

```
if immed_5 == 0
    C Flag = Rm[31]
    if Rm[31] == 0 then
        Rd = 0
    else /* Rm[31] == 1 */]
        Rd = 0xFFFFFFFF
else /* immed_5 > 0 */
    C Flag = Rm[immed_5 - 1]
    Rd = Rm Arithmetic_Shift_Right immed_5
N Flag = Rd[31]
Z Flag = if Rd == 0 then 1 else 0
V Flag = unaffected
```

Equivalent ARM syntax and encoding

```
MOVS  <Rd>, <Rm>, ASR #<immed_5>
```

31 30 29 28	27 26 25 24 23 22 21 20	19 16	15 12	11 7	6 5 4	3 0
1 1 1 0	0 0 0 1 1 0 1 1	SBZ	Rd	immed_5	1 0 0	Rm

ARM DDI 0100F-02
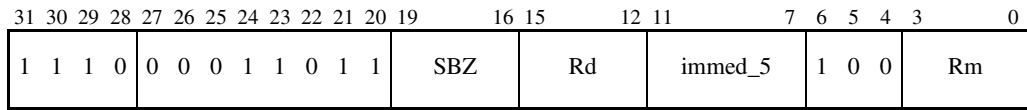
A7.1.12 ASR (2)

15	14	13	12	11	10	9	8	7	6	5		3	2		0
0	1	0	0	0	0	0	1	0	0		Rs			Rd	

ASR (2) provides the signed value of the contents of a register divided by a variable power of 2.

It updates the condition code flags, based on the result.

Syntax

```
ASR  <Rd>, <Rs>
```

where:

<Rd>   Contains the value to be shifted, and is also the destination register for the completed operation.

<Rs>   Specifies the register that contains the value of the shift.

Architecture version

All T variants.

Exceptions

None.

Operation

```
if Rs[7:0] == 0 then
    C Flag = unaffected
    Rd = unaffected
else if Rs[7:0] < 32 then
    C Flag = Rd[Rs[7:0] - 1]
    Rd = Rd Arithmetic_Shift_Right Rs[7:0]
else /* Rs[7:0] >= 32 */
    C Flag = Rd[31]
    if Rd[31] == 0 then
        Rd = 0
    else /* Rd[31] == 1 */
        Rd = 0xFFFFFFFF
N Flag = Rd[31]
Z Flag = if Rd == 0 then 1 else 0
V Flag = unaffected
```

Equivalent ARM syntax and encoding

```
MOVS  <Rd>, <Rd>, ASR <Rs>
```

31 30 29 28	27 26 25 24 23 22 21 20	19 16	15 12	11 8	7 6 5 4	3 0
1 1 1 0	0 0 0 1 1 0 1 1	SBZ	Rd	Rs	0 1 0 1	Rd

ARM DDI 0100F-02

A7.1.13 B (1)

15	14	13	12	11 8	7 0
1	1	0	1	cond	signed_immed_8
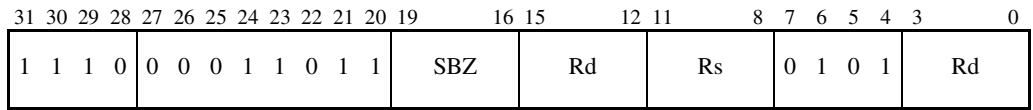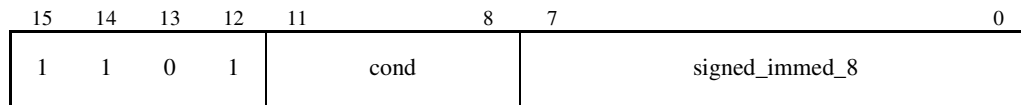
B (1) (Branch) provides a conditional branch to a target address.

Syntax

```
B<cond>  <target_address>
```

where:

<cond>   Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-7.

<target_address>

Specifies the address to branch to. The branch target address is calculated by:

1.   Shifting the 8-bit signed offset field of the instruction left by one bit.
2.   Sign-extending the result to 32 bits.
3.   Adding this to the contents of the PC (which contains the address of the branch instruction plus 4).

The instruction can therefore specify a branch of –256 to +254 bytes.

Architecture version

All T variants.

Exceptions

None.

Operation

```
if ConditionPassed(cond) then
    PC = PC + (SignExtend(signed_immed_8) << 1)
```

Usage

To calculate the correct value of signed_immed_8, the assembler (or other toolkit component) must:

1. Form the base address for the branch. This is the address of the branch instruction, plus 4. In other words, the base address is equal to the PC value read by that instruction.

2. Subtract the base address from the target address to form a byte offset. This offset is always even, because all Thumb instructions are halfword-aligned.

3. If the byte offset is outside the range -256 to +254, use an alternative code-generation strategy or produce an error as appropriate.

4. Otherwise, set the signed_immed_8 field of the instruction to the byte offset divided by 2.

Notes

Memory bounds   Branching backwards past location zero and forwards over the end of the 32-bit address space is UNPREDICTABLE.

AL condition   If the condition field indicates AL (0b1110), the instruction is instead UNDEFINED. When an unconditional branch is required, use the unconditional Branch instruction described in *B (2)* on page A7-21.

NV condition   If the condition field indicates NV (0b1111), the instruction is a SWI instead (see *SWI* on page A7-116).

Equivalent ARM syntax and encoding
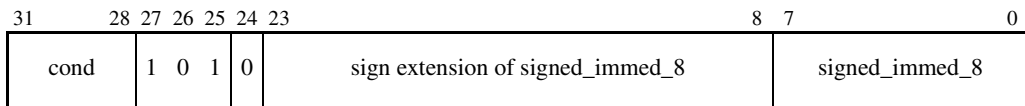
A close equivalent is:

```
B<cond>   <target_address>
```

31	28	27	26	25	24	23		8	7		0
cond		1	0	1	0	sign extension of signed_immed_8			signed_immed_8		

This differs from the Thumb instruction, because the offset in the ARM instruction is shifted left by 2 before being added to the PC, whereas the offset in the Thumb instruction is shifted left by 1. Also, the PC values read by the ARM and Thumb instructions are different.

A7.1.14 B (2)

15	14	13	12	11	10	0
1	1	1	0	0	signed_immed_11	
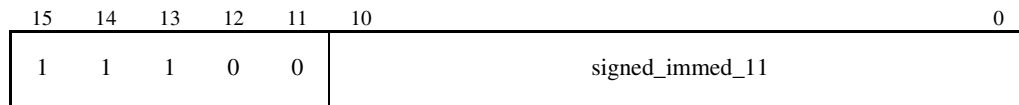
B (2) provides an unconditional branch to a target address.

Syntax

```
B  <target_address>
```

where:

```
<target_address>
```

> Specifies the address to branch to. The branch target address is calculated by:
> 1. Shifting the 11-bit signed offset of the instruction left one bit.
> 2. Sign-extending the result to 32 bits.
> 3. Adding this to the contents of the PC (which contains the address of the branch instruction plus 4).
>
> The instruction can therefore specify a branch of –2048 to +2046 bytes.

Architecture version

All T variants.

Exceptions

None.

Operation

```
PC = PC + (SignExtend(signed_immed_11) << 1)
```

Usage

To calculate the correct value of signed_immed_11, the assembler (or other toolkit component) must:

1. Form the base address for the branch. This is the address of the branch instruction, plus 4. In other words, the base address is equal to the PC value read by that instruction.
2. Subtract the base address from the target address to form a byte offset. This offset is always even, because all Thumb instructions are halfword-aligned.
3. If the byte offset is outside the range -2048 to +2046, use an alternative code-generation strategy or produce an error as appropriate.
4. Otherwise, set the signed_immed_11 field of the instruction to the byte offset divided by 2.

Notes

Memory bounds Branching backwards past location zero and forwards over the end of the 32-bit address space is UNPREDICTABLE.

Equivalent ARM syntax and encoding
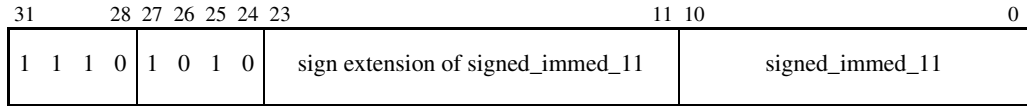
A close equivalent is:

```
B   <target_address>
```

31 28	27 26 25 24	23 11	10 0
1 1 1 0	1 0 1 0	sign extension of signed_immed_11	signed_immed_11

This differs from the Thumb instruction, because the offset in the ARM instruction is shifted left by 2 before being added to the PC, whereas the offset in the Thumb instruction is shifted left by 1. Also, the PC values read by the ARM and Thumb instructions are different.

A7.1.15 BIC

15	14	13	12	11	10	9	8	7	6	5		3	2		0
0	1	0	0	0	0	1	1	1	0		Rm			Rd	

BIC (Bit Clear) performs a bitwise AND of the value of one register and the bitwise inverse of the value of another register.

BIC updates the condition code flags, based on the result.

Syntax

```
BIC  <Rd>, <Rm>
```

where:

<Rd>	Is the register containing the value to be ANDed, and is also the destination register for the completed operation.
<Rm>	Specifies the register that contains the value whose complement is ANDed with the value in <Rd>.

Architecture version

All T variants.

Exceptions

None.

Operation

```
Rd = Rd AND NOT Rm
N Flag = Rd[31]
Z Flag = if Rd == 0 then 1 else 0
C Flag = unaffected
V Flag = unaffected
```

Equivalent ARM syntax and encoding

```
BICS  <Rd>, <Rd>, <Rm>
```

31	30	29	28	27	26	25	24	23	22	21	20	19		16	15		12	11	10	9	8	7	6	5	4	3		0
1	1	1	0	0	0	0	1	1	1	0	1		Rd			Rd		0	0	0	0	0	0	0	0		Rm	
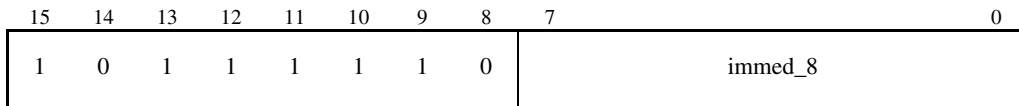
Copyright © 1996–2003 ARM Limited. All rights reserved.
Confidential - Draft

A7.1.16 BKPT

15	14	13	12	11	10	9	8	7	0
1	0	1	1	1	1	1	0	immed_8	

BKPT (Breakpoint) causes a software breakpoint to occur. This breakpoint can be handled by an exception handler installed on the Prefetch Abort vector. In implementations which also include debug hardware, the hardware can optionally override this behavior and handle the breakpoint itself. When this occurs, the Prefetch Abort vector is not entered.

Syntax

```
BKPT   <immed_8>
```

where:

<immed_8>   Is an 8-bit immediate value, which is placed in bits[7:0] of the instruction. This value is ignored by the ARM hardware, but can be used by a debugger to store additional information about the breakpoint.

Architecture version

T variants of version 5 and above.

Exceptions

Prefetch Abort.

Operation

```
if (not overridden by debug hardware)
    R14_abt   = address of BKPT instruction + 4
    SPSR_abt  = CPSR
    CPSR[4:0] = 0b10111               /* Enter Abort mode */
    CPSR[5]   = 0                     /* Execute in ARM state */
    /* CPSR[6] is unchanged */
    CPSR[7]   = 1                     /* Disable normal interrupts */
    if high vectors configured then
        PC    = 0xFFFF000C
    else
        PC    = 0x0000000C
```

Usage

The exact usage of BKPT depends on the debug system being used. A debug system can use BKPT in two ways:

• Debug hardware (if present) does not override the normal behavior of BKPT, and so the Prefetch Abort vector is entered. If the system also allows real Prefetch Aborts to occur, the Prefetch Abort handler determines (in a system-dependent manner) whether the vector entry occurred as a result of a BKPT instruction or as a result of a real Prefetch Abort, and branches to debug code or Prefetch Abort code accordingly. Otherwise, the Prefetch Abort handler just branches straight to debug code.

When used in this manner, BKPT must be avoided within abort handlers, as it corrupts R14_abt and SPSR_abt. For the same reason, it must also be avoided within FIQ handlers, as an FIQ interrupt can occur within an abort handler.

• Debug hardware overrides the normal behavior of BKPT and handles the software breakpoint itself. When finished, it typically either resumes execution at the instruction following the BKPT, or replaces the BKPT in memory with another instruction and resumes execution at that instruction.

When BKPT is used in this manner, R14_abt and SPSR_abt are not corrupted, and so the above restrictions about its use in abort and FIQ handlers do not apply.

Notes

Hardware override Debug hardware in an implementation is specifically permitted to override the normal behavior of BKPT. Because of this, software must not use this instruction for purposes other than those permitted by the debug system being used (if any). In particular, software cannot rely on the Prefetch Abort exception occurring, unless either there is guaranteed to be no debug hardware in the system or the debug system specifies that it occurs.

For more information, consult the documentation for the debug system being used.

Equivalent ARM syntax and encoding

```
BKPT    <immed_8>
```

31 30 29 28	27 26 25 24 23 22 21 10	19 18 17 6 15 14 13 12	11 8	7 4	3 0
1 1 1 0	0 0 0 1 0 0 1 0	0 0 0 0 0 0 0 0	immed_8 [7:4]	0 1 1 1	immed_8 [3:0]

Confidential - Draft

A7.1.17 BL, BLX (1)

15	14	13	12	11	10	0
1	1	1	H		offset_11	
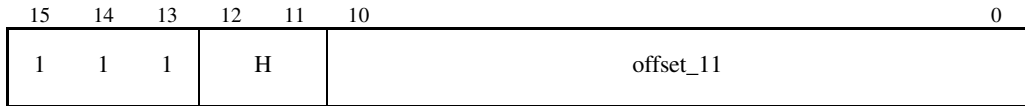
BL (Branch with Link) provides an unconditional subroutine call to another Thumb routine. The return from subroutine is typically performed by one of the following:

* MOV PC,LR
* BX LR
* a POP instruction that loads the PC.

BLX (1) (Branch with Link and Exchange instruction set) provides an unconditional subroutine call to an ARM routine. The return from subroutine is typically performed by a BX LR instruction, or an LDR or LDM instruction that loads the PC.

To allow for a reasonably large offset to the target subroutine, each of these two instructions is automatically translated by the assembler into a sequence of two 16-bit Thumb instructions:

* The first Thumb instruction has H == 10 and supplies the high part of the branch offset. This instruction sets up for the subroutine call and is shared between the BL and BLX forms.

* The second Thumb instruction has H == 11 (for BL) or H == 01 (for BLX). It supplies the low part of the branch offset and causes the subroutine call to take place.

Syntax

```
BL   <target_addr>
BLX  <target_addr>
```

where:

<target_addr>   Specifies the address to branch to. The branch target address is calculated by:
1. Shifting the offset_11 field of the first instruction left twelve bits.
2. Sign-extending the result to 32 bits.
3. Adding this to the contents of the PC (which contains the address of the first instruction plus 4).
4. Adding twice the offset_11 field of the second instruction. For BLX, the resulting address is forced to be word-aligned by clearing bit[1].

The instruction can therefore specify a branch of approximately ±4MB, see *Usage* on page A7-27 for the exact range.

Architecture version

BL (H == 10 and H == 11 forms) is in all T variants.

BLX (H == 01 form) is in T variants of version 5 and above.

Exceptions

None.

Operation

```
if H == 10 then
    LR = PC + (SignExtend(offset_11) << 12)

else if H == 11 then
    PC = LR + (offset_11 << 1)
    LR = (address of next instruction) | 1

else if H == 01 then
    PC = (LR + (offset_11 << 1)) AND 0xFFFFFFFC
    LR = (address of next instruction) | 1
    CPSR T bit = 0
```

Usage

To generate the correct pair of instructions, the assembler (or other toolkit component) must first generate the branch offset, as follows:

1. Form the base address for the branch. This is the address of the first of the two Thumb instructions (the one with H == 10), plus 4. In other words, the base address is equal to the PC value read by that instruction.

2. If the instruction is BLX, set bit[1] of the target address to be equal to bit[1] of the base address. This is an exception to the normal rule that bits[1:0] of the address of an ARM instruction are 0b00. This adjustment is required to ensure that the restrictions associated with the H == 01 form of the instruction are obeyed.

3. Subtract the base address from the target address to form the offset.

The resulting offset is always even. If the offset lies outside the range:

```
    -2^22 <= offset <= +2^22 - 2
```

the target address lies outside the addressing range of these instructions. This results in alternative code or an error, as appropriate.

If the offset is in range, a sequence of two Thumb instructions must be generated, both using the above form:

- The first with H == 10 and offset_11 = offset[22:12].
- The second with H == 11 (for BL) or H== 01 (for BLX) and offset_11 = offset[11:1].

Notes

Encoding   If H == 00, the instruction is an unconditional branch instruction instead (see the Thumb instruction *B (2)* on page A7-21).

Bit[0] for BLX   If H == 01, then bit[0] of the instruction must be zero, or the instruction is UNDEFINED. The offset calculation method described in *Usage* above ensures that the offset calculated for a BLX instruction is a multiple of four, and that this restriction is obeyed.

Memory bounds   Branching backwards past location zero and forwards over the end of the 32-bit address space is UNPREDICTABLE.

Instruction pairs   These Thumb instructions must always occur in the pairs described above. Specifically:

- If a Thumb instruction at address A is the H==10 form of this instruction, the Thumb instruction at address A+2 must be either the H==01 or the H==11 form of this instruction.

- If a Thumb instruction at address A is either the H==01 or the H==11 form of this instruction, the Thumb instruction at address A-2 must be the H==10 form of this instruction.

Also, except as noted below under *Exceptions*, the second instruction of the pair must not be the target of any branch, whether as the result of a branch instruction or of some other instruction that changes the PC.

Failure to adhere to any of these restrictions can result in UNPREDICTABLE behavior.

Exceptions   It is IMPLEMENTATION DEFINED whether processor exceptions can occur between the two instructions of a BL or BLX pair. If they can, the ARM instructions designed for use for exception returns must be capable of returning correctly to the second instruction of the pair. So, exception handlers need take no special precautions about returning to the second instruction of a BL or BLX pair.

Equivalent ARM syntax and encoding

Close equivalents to these instruction pairs are as follows.

To call a Thumb subroutine:

```
BLX   <target_addr>
```

31 30 29 28	27 26 25 24	23	22 21 20 0
1 1 1 1	1 0 1 L	offset sign	offset[22:2]
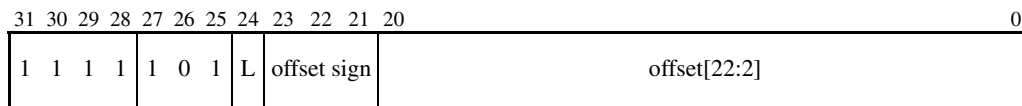
where L == offset[1].

To call an ARM routine:

```
BL   <target_addr>
```

31 30 29 28	27 26 25 24	23	22 21 20 0
1 1 1 0	1 0 1 1	offset sign	offset[22:2]
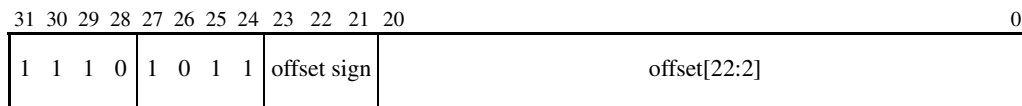
These differ slightly from the Thumb instruction pairs because of the different values of the PC in ARM and Thumb code. This can be compensated for by adjusting the offset by 4.

A7.1.18 BLX (2)

15	14	13	12	11	10	9	8	7	6	5		3	2		0
0	1	0	0	0	1	1	1	1	H2	Rm			SBZ		

BLX (2) calls an ARM or Thumb subroutine from the Thumb instruction set, at an address specified in a register. This instruction branches and selects the instruction decoder to use to decode the instructions at the branch destination.

The CPSR T bit is updated with bit[0] of the value of register Rm. To return from the subroutine to the caller, use BX R14.

Syntax

BLX   <Rm>

where:

<Rm>   Is the register that contains the branch target address. It can be any of R0 to R14. The register number is encoded in the instruction in H2 (most significant bit) and Rm (remaining three bits). If R15 is specified for <Rm>, the results are UNPREDICTABLE.

Architecture version

T variants of version 5 and above.

Exceptions

None.

Operation

```
LR = (address of the instruction after this BLX) | 1
CPSR T bit = Rm[0]
PC = Rm[31:1] << 1
```

Notes

Encoding   Bit 7 is the H1 bit for some of the other instructions that access the high registers. If it is 0 for this instruction, rather than 1 as shown, the instruction is a BX instruction instead (see *BX* on page A7-32).

ARM/Thumb state transfers

If Rm[1:0] == 0b10, the result is UNPREDICTABLE, as branches to non word-aligned addresses are impossible in ARM state.

Equivalent ARM syntax and encoding

```
BLX   <Rm>
```

31 30 29 28	27 26 25 24	23 22 21 20	19 16	15 12	11 8	7 6 5 4	3	2 0
1 1 1 0	0 0 0 1	0 0 1 0	SBO	SBO	SBO	0 0 1 1	H2	Rm

Copyright © 1996–2003 ARM Limited. All rights reserved.
Confidential - Draft

A7.1.19 BX

15	14	13	12	11	10	9	8	7	6	5		3	2		0
0	1	0	0	0	1	1	1	0	H2	Rm			SBZ		

BX (Branch and Exchange) branches between ARM code and Thumb code.

Syntax

```
BX   <Rm>
```

where:

<Rm>   Is the register that contains the branch target address. It can be any of R0 to R15. The register
                number is encoded in the instruction in H2 (most significant bit) and Rm (remaining three
                bits).

Architecture version

All T variants.

Exceptions

None.

Operation

```
CPSR T bit = Rm[0]
PC = Rm[31:1] << 1
```

Usage

The normal subroutine return instruction in Thumb code is `BX  R14`. The following subroutine call
instructions leave a suitable return value in R14:

•   ARM `BLX` instructions (See *BLX (1)* on page A4-16 and *BLX (2)* on page A4-18)

•   Thumb `BL` and `BLX` instructions (see *BL, BLX (1)* on page A7-26 and *BLX (2)* on page A7-30).

In T variants of ARM architecture version 4, a subroutine call to an ARM routine can be performed by a
code sequence of the form:

```
<Put address of routine to call in Ra>
MOV   LR,PC       ; Return to second following instruction
BX    Ra
```

In T variants of ARM architecture 5 and above, a subroutine call to an ARM routine can be performed more
efficiently with a `BLX` instruction (see *BL, BLX (1)* on page A7-26 and *BLX (2)* on page A7-30).

Notes

Encoding   Bit 7 is the H1 bit for some of the other instructions that access the high registers. If it is 1 for this instruction, rather than 0 as shown, the instruction is:

- a BLX instruction instead in ARM architecture version 5 and above (see *BLX (2)* on page A7-30)

- UNPREDICTABLE prior to ARM architecture version 5.

ARM/Thumb state transfers

If Rm[1:0] == 0b10, the result is UNPREDICTABLE, as branches to non word-aligned addresses are impossible in ARM state.

Use of R15   Register 15 can be specified for <Rm>. If this is done, R15 is read as normal for Thumb code, that is, it is the address of the BX instruction itself plus 4. If the BX instruction is at a word-aligned address, this results in a branch to the next word, executing in ARM state. However, if the BX instruction is not at a word-aligned address, this means that the results of the instruction are UNPREDICTABLE (because the value read for R15 has bits[1:0]==0b10).

Equivalent ARM syntax and encoding

A close equivalent is:

```
BX    <Rm>
```

31 30 29 28	27 26 25 24	23 22 21 20	19 16	15 12	11 8	7 6 5 4	3	0
1 1 1 0	0 0 0 1	0 0 1 0	SBO	SBO	SBO	0 0 0 1	H2	Rm

This ARM instruction is not quite equivalent to the Thumb instruction, because their specified behavior differs when <Rm> is R15.

A7.1.20 CMN

15	14	13	12	11	10	9	8	7	6	5	3	2	0
0	1	0	0	0	0	1	0	1	1	Rm		Rn	

CMN (Compare Negative) compares a register value with the negation of another register value. The condition flags are updated, based on the result of adding the two register values, so that subsequent instructions can be conditionally executed (using a conditional branch).

Syntax

CMN   <Rn>, <Rm>

where:

<Rn>   Is the register containing the first value for comparison.

<Rm>   Is the register containing the second value for comparison.

Architecture version

All T variants.

Exceptions

None.

Operation

```
alu_out = Rn + Rm
N Flag = alu_out[31]
Z Flag = if alu_out == 0 then 1 else 0
C Flag = NOT BorrowFrom(Rn + Rm)
V Flag = OverflowFrom(Rn + Rm)
```

Equivalent ARM syntax and encoding

CMN   <Rn>, <Rm>

31	30	29	28	27	26	25	24	23	22	21	20	19	16	15	12	11	10	9	8	7	6	5	4	3	0
1	1	1	0	0	0	0	1	0	1	1	1	Rn		SBZ		0	0	0	0	0	0	0	0	Rm	

A7.1.21 CMP (1)

15	14	13	12	11	10	8	7	0
0	0	1	0	1		Rn		immed_8
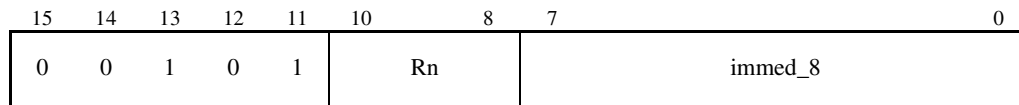
CMP (1) (Compare) compares a register value with a large immediate value. The condition flags are updated, based on the result of subtracting the constant from the register value, so that subsequent instructions can be conditionally executed (using a conditional branch).

Syntax

```
CMP  <Rn>, #<immed_8>
```

where:

<Rn>   Is the register containing the first value for comparison.

<immed_8>   Is the 8-bit second value for comparison.

Architecture version

All T variants.

Exceptions

None.

Operation

```
alu_out = Rn - immed_8
N Flag = alu_out[31]
Z Flag = if alu_out == 0 then 1 else 0
C Flag = NOT BorrowFrom(Rn - immed_8)
V Flag = OverflowFrom(Rn - immed_8)
```

Equivalent ARM syntax and encoding

```
CMP  <Rn>, #<immed_8>
```

31 30 29 28	27 26 25 24 23 22 21 20	19 16	15 12	11 10 9 8	7 0
1 1 1 0	0 0 1 1 0 1 0 1	Rn	SBZ	0 0 0 0	immed_8

Copyright © 1996–2003 ARM Limited. All rights reserved.
Confidential - Draft
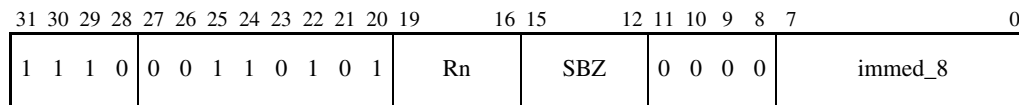
A7.1.22 CMP (2)

15	14	13	12	11	10	9	8	7	6	5		3	2		0
0	1	0	0	0	0	1	0	1	0		Rm			Rn	

CMP (2) compares two register values. The condition code flags are updated, based on the result of subtracting the second register value from the first, so that subsequent instructions can be conditionally executed (using a conditional branch).

Syntax

CMP   <Rn>, <Rm>

where:

<Rn>   Is the register containing the first value for comparison.

<Rm>   Is the register containing the second value for comparison.

Architecture version

All T variants.

Exceptions

None.

Operation

```
alu_out = Rn - Rm
N Flag = alu_out[31]
Z Flag = if alu_out == 0 then 1 else 0
C Flag = NOT BorrowFrom(Rn - Rm)
V Flag = OverflowFrom(Rn - Rm)
```

Equivalent ARM syntax and encoding

CMP   <Rn>, <Rm>

31	30	29	28	27	26	25	24	23	22	21	20	19		16	15		12	11	10	9	8	7	6	5	4	3		0
1	1	1	0	0	0	0	1	0	1	0	1		Rn			SBZ		0	0	0	0	0	0	0	0		Rm	

A7.1.23 CMP (3)

15	14	13	12	11	10	9	8	7	6	5		3	2		0
0	1	0	0	0	1	0	1	H1	H2	Rm			Rn		

CMP (3) compares the values of two registers, one or both of which are high registers. The condition flags are updated, based on the result of subtracting the second register value from the first, so that subsequent instructions can be conditionally executed (using a conditional branch).

Syntax

CMP <Rn>, <Rm>

where:

<Rn>	Is the register containing the first value. It can be any of R0 to R15. Its number is encoded in the instruction in H1 (most significant bit) and Rn (remaining three bits).
<Rm>	Is the register containing the second value. It can be any of R0 to R15. Its number is encoded in the instruction in H2 (most significant bit) and Rm (remaining three bits).

Architecture version

All T variants.

Exceptions

None.

Operation

```
alu_out = Rn - Rm
N Flag = alu_out[31]
Z Flag = if alu_out == 0 then 1 else 0
C Flag = NOT BorrowFrom(Rn - Rm)
V Flag = OverflowFrom(Rn - Rm)
```

Notes

Operand restriction   If a low register is specified for both <Rn> and <Rm> (H1==0 and H2==0), the result is UNPREDICTABLE.

Copyright © 1996–2003 ARM Limited. All rights reserved.
Confidential - Draft

Equivalent ARM syntax and encoding

A close equivalent is:

```
CMP   <Rn>, <Rm>
```

31 30 29 28	27 26 25 24 23 22 21 20	19	18 16	15 12	11 10 9 8 7 6 5 4	3	2 0
1 1 1 0	0 0 0 1 0 1 0 1	H1	Rn	SBZ	0 0 0 0 0 0 0 0	H2	Rm
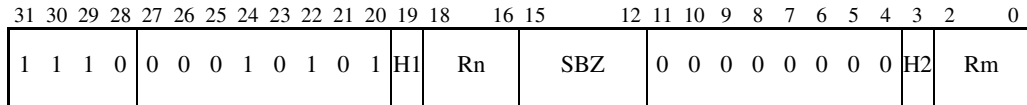
There are slight differences when the instruction accesses the PC, because of the different definitions of the PC when executing ARM and Thumb code.

A7.1.24 CPS

15			12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	0	1	1	0	0	1	1	imod	0	A	I	F

CPS (Change Processor State) changes one or more of the A, I, and F bits of the CPSR, without changing other CPSR bits.

Syntax

```
CPS<effect> <iflags>
```

where:

<effect> Specifies what effect is wanted on the interrupt disable bits A, I, and F in the CPSR. This is either:

    IE   Interrupt Enable, encoded by imod == 0b10. This sets the specified bits to 0.

    ID   Interrupt Disable, encoded by imod == 0b11. This sets the specified bits to 1.

<iflags> Is a sequence of one or more of the following, specifying which interrupt disable flags are affected:

    a   Sets the A bit (bit[8]), causing the specified effect on the CPSR A bit.

    i   Sets the I bit (bit[7]), causing the specified effect on the CPSR A bit.

    f   Sets the F bit (bit[6]), causing the specified effect on the CPSR A bit.

Architecture version

T variants of version 6 and above.

Exceptions

None.

Operation

```
if InAPrivilegedMode() then
    if A == 1 then CPSR[8] = imod[0]
    if I == 1 then CPSR[7] = imod[0]
    if F == 1 then CPSR[6] = imod[0]
/* else no change to interrupt disable bits */
```

Notes

User mode   This instruction has no effect in User mode.

Equivalent ARM syntax and encoding

```
CPS  <effect>, <flags>
```

31 30 29 28	27 26 25 24 23 22 21 20	19 18	17	16	15 9	8	7	6	5	4 0
1 1 1 1	0 0 0 1 0 0 0 0	imod	0	0	SBZ	A	I	F	0	SBZ

ARM DDI 0100F-02
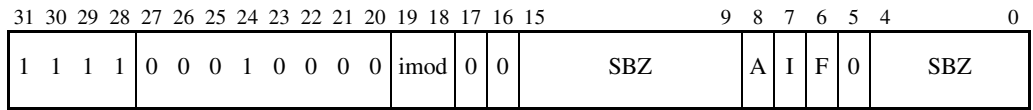
A7.1.25 CPY

15	14	13	12	11	10	9	8	7	6	5	3	2	0
0	1	0	0	0	1	1	0	H1	H2	Rm		Rd	

CPY (Copy) moves a value from one high or low register to another high or low register, without changing the flags.

Syntax

```
CPY  <Rd>, <Rm>
```

where:

<Rd>   Is the destination register for the operation. It can be any of R0 to R15, and its number is encoded in the instruction in H1 (most significant bit) and Rd (remaining three bits).

<Rm>   Is the register containing the value to be copied. It can be any of R0 to R15, and its number is encoded in the instruction in H2 (most significant bit) and Rm (remaining three bits).

Architecture version

T variants of version 6 and above.

Exceptions

None.

Operation

```
Rd = Rm
```

Usage

CPY PC,R14 can be used as a subroutine return instruction if it is known that the caller is also a Thumb routine. However, it is more usual to use BX R14 (see *BX* on page A7-32), which works regardless of whether the caller is an ARM routine or a Thumb routine.

Notes

Encoding   CPY has the same functionality as *MOV (3)* on page A7-74, and uses the same instruction encoding, but has an assembler syntax that allows both operands to be low registers.

Equivalent ARM syntax and encoding

A close equivalent is:

```
CPY   <Rd>, <Rm>
```
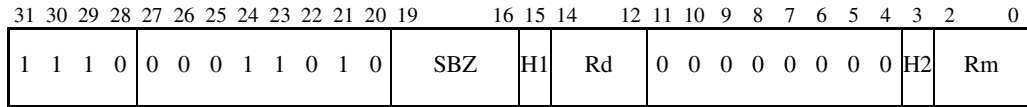
31 30 29 28	27 26 25 24 23 22 21 20	19 16	15	14 12	11 10 9 8 7 6 5 4	3	2 0
1 1 1 0	0 0 0 1 1 0 1 0	SBZ	H1	Rd	0 0 0 0 0 0 0 0	H2	Rm

There are slight differences when the instruction accesses the PC, because of the different definitions of the PC when executing ARM and THUMB code.

A7.1.26 EOR

15	14	13	12	11	10	9	8	7	6	5	3	2	0
0	1	0	0	0	0	0	0	0	1	Rm		Rd	

EOR (Exclusive OR) performs a bitwise EOR of the values from two registers.

EOR updates the condition code flags, based on the result.

Syntax

```
EOR   <Rd>, <Rm>
```

where:

<Rd>   Specifies the register containing the first operand, and is also the destination register.

<Rm>   Specifies the register containing the second operand.

Architecture version

All T variants.
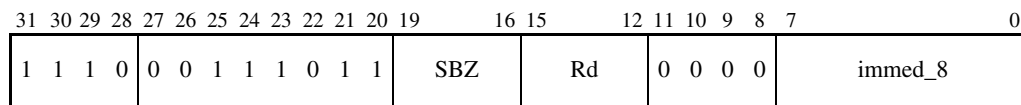
Exceptions

None.

Operation

```
Rd = Rd EOR Rm
N Flag = Rd[31]
Z Flag = if Rd == 0 then 1 else 0
C Flag = unaffected
V Flag = unaffected
```

Equivalent ARM syntax and encoding

```
EORS   <Rd>, <Rd>, <Rm>
```

31 30 29 28	27 26 25 24 23 22 21 20	19 16	15 12	11 10 9 8 7 6 5 4	3 0
1 1 1 0	0 0 0 0 0 0 1 1	Rd	Rd	0 0 0 0 0 0 0 0	Rm

Copyright © 1996–2003 ARM Limited. All rights reserved.
Confidential - Draft

A7.1.27 LDMIA

15	14	13	12	11	10	8	7	0
1	1	0	0	1		Rn	register_list	
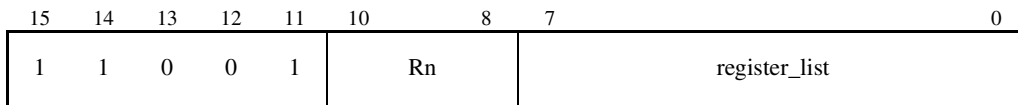
LDMIA (Load Multiple Increment After) loads a non-empty subset, or possibly all, of the general-purpose registers from sequential memory locations.

Syntax

```
LDMIA   <Rn>!, <registers>
```

where:

<Rn>	Is the register containing the start address for the instruction.
!	Causes base register write-back, and is not optional.
<registers>	Is a list of registers to be loaded, separated by commas and surrounded by { and }. The list is encoded in the register_list field of the instruction, by setting bit[i] to 1 if register Ri is included in the list and to 0 otherwise, for each of i=0 to 7.
	At least one register must be loaded. If bits[7:0] are all zero, the result is UNPREDICTABLE.
	The registers are loaded in sequence, the lowest-numbered register from the lowest memory address (start_address), through to the highest-numbered register from the highest memory address (end_address).
	The start_address is the value of the base register <Rn>. Subsequent addresses are formed by incrementing the previous address by four. One address is produced for each register that is specified in <registers>.
	The end_address value is four less than the sum of the value of the base register and four times the number of registers specified in <registers>.
	Finally, the base register <Rn> is incremented by four times the numbers of registers in <registers>.

Architecture version

All T variants.

Exceptions

Data Abort.

Operation

```
start_address = Rn
end_address = Rn + (Number_Of_Set_Bits_In(register_list) * 4) - 4
address = start_address
for i = 0 to 7
    if register_list[i] == 1
        Ri = Memory[address,4]
        address = address + 4
assert end_address == address - 4
Rn = Rn + (Number_Of_Set_Bits_In(register_list) * 4)
```

Usage

Use LDMIA as a block load instruction. Combined with STMIA (Store Multiple), it allows efficient block copy.

Notes

Operand restrictions

If the base register <Rn> is specified in <registers>, the final value of <Rn> is the loaded value (not the written-back value).

Data Abort   For details of the effects of the instruction if a Data Abort occurs, see *Effects of data-aborted instructions* on page A2-21.

Alignment   Load Multiple instructions ignore the least significant two bits of address.

If an implementation includes a System Control coprocessor (see Chapter B3 *The System Control Coprocessor*) and alignment checking is enabled, an address with bits[1:0] != 0b00 causes an alignment exception.

Time order   The time order of the accesses to individual words of memory generated by this instruction is only defined in some circumstances. See *Data accesses to memory-mapped I/O* on page B2-27 for details.

Equivalent ARM syntax and encoding

If <Rn> is not in the register list (W == 1):

```
LDMIA   <Rn>!, <registers>
```

If <Rn> is in the register list (W == 0):

```
LDMIA   <Rn>, <registers>
```

31 30 29 28	27 26 25 24 23 22	21	20	19 16	15 14 13 12 11 10 9 8	7 0
1 1 1 0	1 0 0 0 1 0	W	1	Rn	0 0 0 0 0 0 0 0	register_list

Copyright © 1996–2003 ARM Limited. All rights reserved.
Confidential - Draft

A7.1.28 LDR (1)

15	14	13	12	11	10			6	5		3	2		0
0	1	1	0	1		immed_5				Rn			Rd	
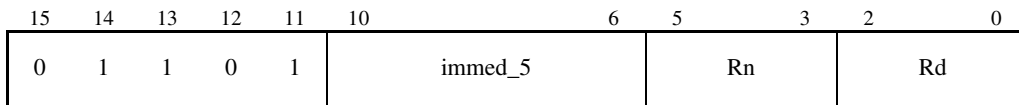
LDR (1) (Load Register) allows 32-bit memory data to be loaded into a general-purpose register.
The addressing mode is useful for accessing structure (record) fields. With an offset of zero, the address
produced is the unaltered value of the base register <Rn>.

Syntax

```
LDR  <Rd>, [<Rn>, #<immed_5> * 4]
```

where:

<Rd>   Is the destination register for the word loaded from memory.

<Rn>   Is the register containing the base address for the instruction.

<immed_5>   Is a 5-bit value that is multiplied by 4 and added to the value of <Rn> to form the memory
            address.

Architecture version

All T variants.

Exceptions

Data Abort.

Operation

```
address = Rn + (immed_5 * 4)
if address[1:0] == 0b00
    data = Memory[address,4]
else
    data = UNPREDICTABLE
Rd = data
```

ARM DDI 0100F-02

Notes

Data Abort   For details of the effects of the instruction if a Data Abort occurs, see *Effects of data-aborted instructions* on page A2-21.

Alignment   If the memory address is not word-aligned and no Data Abort occurs, the value written to the destination register is UNPREDICTABLE.

If an implementation includes a System Control coprocessor (see Chapter B3 *The System Control Coprocessor*) and alignment checking is enabled, an address with bits[1:0] != 0b00 causes an alignment exception (a type of Data Abort).

Equivalent ARM syntax and encoding

```
LDR  <Rd>, [<Rn>, #<immed_5> * 4]
```

31 30 29 28	27 26 25 24 23 22 21 20	19 16	15 12	11 10 9 8 7	6 2	1 0
1 1 1 0	0 1 0 1 1 0 0 1	Rn	Rd	0 0 0 0 0	immed_5	0 0

A7.1.29 LDR (2)

15	14	13	12	11	10	9	8	6	5	3	2	0
0	1	0	1	1	0	0	Rm		Rn		Rd	
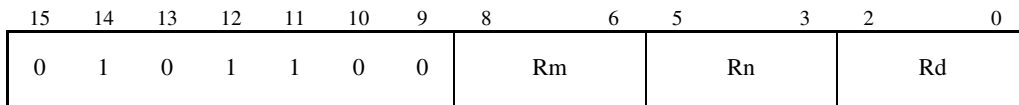
LDR (2) loads 32-bit memory data into a general-purpose register. The addressing mode is useful for pointer+large offset arithmetic and for accessing a single element of an array.

Syntax

```
LDR  <Rd>, [<Rn>, <Rm>]
```

where:

<Rd>   Is the destination register for the word loaded from memory.

<Rn>   Is the register containing the first value used in forming the memory address.

<Rm>   Is the register containing the second value used in forming the memory address.

Architecture version

All T variants.

Exceptions

Data Abort.

Operation

```
address = Rn + Rm
if address[1:0] == 0b00
    data = Memory[address,4]
else
    data = UNPREDICTABLE
Rd = data
```

Notes

Data Abort   For details of the effects of the instruction if a Data Abort occurs, see *Effects of data-aborted instructions* on page A2-21.

Alignment   If the memory address is not word-aligned and no Data Abort occurs, the value written to the destination register is UNPREDICTABLE.

If an implementation includes a System Control coprocessor (see Chapter B3 *The System Control Coprocessor*) and alignment checking is enabled, an address with bits[1:0] != 0b00 causes an alignment exception (a type of Data Abort).

Equivalent ARM syntax and encoding

```
LDR  <Rd>, [<Rn>, <Rm>]
```

31 30 29 28	27 26 25 24 23 22 21 20	19 16	15 12	11 10 9 8 7 6 5 4	3 0
1 1 1 0	0 1 1 1 1 0 0 1	Rn	Rd	0 0 0 0 0 0 0 0	Rm

A7.1.30 LDR (3)

15	14	13	12	11	10	8	7	0
0	1	0	0	1	Rd		immed_8	
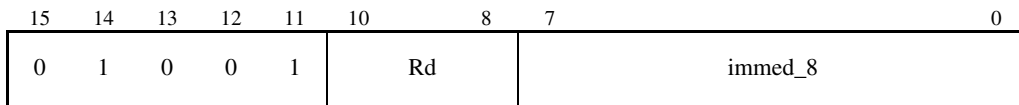
LDR (3) loads 32-bit memory data into a general-purpose register. The addressing mode is useful for accessing PC-relative data.

Syntax

```
LDR  <Rd>, [PC, #<immed_8> * 4]
```

where:

<Rd>   Is the destination register for the word loaded from memory.

PC   Is the program counter. Its value is used to calculate the memory address. Bit 1 of the PC value is forced to zero for the purpose of this calculation, so the address is always word-aligned.

<immed_8> Is an 8-bit value that is multiplied by 4 and added to the value of the PC to form the memory address.

Architecture version

All T variants.

Exceptions

Data Abort.

Operation

```
address = (PC[31:2] << 2) + (immed_8 * 4)
Rd = Memory[address, 4]
```

Notes

Data Abort   For details of the effects of the instruction if a Data Abort occurs, see *Effects of data-aborted instructions* on page A2-21.

Equivalent ARM syntax and encoding

A close equivalent is:

```
LDR  <Rd>, [PC, #<immed_8> * 4]
```

31 30 29 28	27 26 25 24 23 22 21 20	19 18 17 16	15 12	11 10	9 2	1 0
1 1 1 0	0 1 0 1 1 0 0 1	1 1 1 1	Rd	0 0	immed_8	0 0

There are slight differences caused by the different definitions of the PC and the fact that the Thumb instruction ignores bit[1] of the PC.

A7.1.31 LDR (4)

15	14	13	12	11	10 8	7 0
1	0	0	1	1	Rd	immed_8
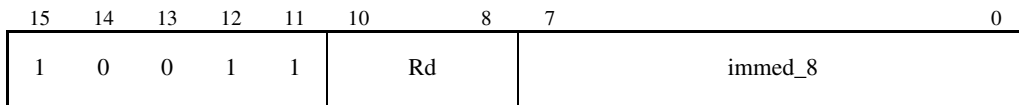
LDR (4) loads 32-bit memory data into a general-purpose register. The addressing mode is useful for accessing stack data.

Syntax

```
LDR  <Rd>, [SP, #<immed_8> * 4]
```

where:

`<Rd>`	Is the destination register for the word loaded from memory.
`SP`	Is the stack pointer. Its value is used to calculate the memory address.
`<immed_8>`	Is an 8-bit value that is multiplied by 4 and added to the value of the SP to form the memory address.

Architecture version

All T variants.

Exceptions

Data Abort.

Operation

```
address = SP + (immed_8 * 4)
if address[1:0] == 0b00
    data = Memory[address,4]
else
    data = UNPREDICTABLE
Rd = data
```

Notes

Data Abort   For details of the effects of the instruction if a Data Abort occurs, see *Effects of data-aborted instructions* on page A2-21.

Alignment   If the memory address is not word-aligned and no Data Abort occurs, the value written to the destination register is UNPREDICTABLE.

If an implementation includes a System Control coprocessor (see Chapter B3 *The System Control Coprocessor*) and alignment checking is enabled, an address with bits[1:0] != 0b00 causes an alignment exception (a type of Data Abort).

Equivalent ARM syntax and encoding

```
LDR  <Rd>, [SP, #<immed_8> * 4]
```

31 30 29 28	27 26 25 24 23 22 21 20	19 18 17 16	15 12	11 10	9 2	1 0
1 1 1 0	0 1 0 1 1 0 0 1	1 1 0 1	Rd	0 0	immed_8	0 0

A7.1.32 LDRB (1)

15	14	13	12	11	10	6	5	3	2	0
0	1	1	1	1	immed_5		Rn		Rd	

`LDRB` (1) (Load Register Byte) loads a byte from memory, zero-extends it to form a 32-bit word, and writes the result to a general-purpose register. The addressing mode is useful for accessing structure (record) fields. With an offset of zero, the address produced is the unaltered value of the base register `<Rn>`.

Syntax

```
LDRB  <Rd>, [<Rn>, #<immed_5>]
```

where:

`<Rd>`	Is the destination register for the byte loaded from memory.
`<Rn>`	Is the register containing the base address for the instruction.
`<immed_5>`	Is a 5-bit value that is added to the value of `<Rn>` to form the memory address.

Architecture version

All T variants.

Exceptions

Data Abort.

Operation

```
address = Rn + immed_5
Rd = Memory[address,1]
```

Notes

Data Abort   For details of the effects of the instruction if a Data Abort occurs, see *Effects of data-aborted instructions* on page A2-21.

Equivalent ARM syntax and encoding

```
LDRB  <Rd>, [<Rn>, #<immed_5>]
```

31 30 29 28	27 26 25 24 23 22 21 20	19 16	15 12	11 10 9 8 7 6 5 4	0
1 1 1 0	0 1 0 1 1 1 0 1	Rn	Rd	0 0 0 0 0 0 0	immed_5

Copyright © 1996–2003 ARM Limited. All rights reserved.   ARM DDI 0100F-02
Confidential - Draft

A7.1.33 LDRB (2)

15	14	13	12	11	10	9	8		6	5		3	2		0
0	1	0	1	1	1	0		Rm			Rn			Rd	

LDRB (2) loads a byte from memory, zero-extends it to form a 32-bit word, and writes the result to a general-purpose register. The addressing mode is useful for pointer+large offset arithmetic and for accessing a single element of an array.

Syntax

```
LDRB  <Rd>, [<Rn>, <Rm>]
```

where:

<Rd>	Is the destination register for the byte loaded from memory.
<Rn>	Is the register containing the first value used in forming the memory address.
<Rm>	Is the register containing the second value used in forming the memory address.

Architecture version

All T variants.

Exceptions

Data Abort.

Operation

```
address = Rn + Rm
Rd = Memory[address,1]
```

Notes

Data Abort For details of the effects of the instruction if a Data Abort occurs, see *Effects of data-aborted instructions* on page A2-21.

Equivalent ARM syntax and encoding

```
LDRB  <Rd>, [<Rn>, <Rm>]
```

31	30	29	28	27	26	25	24	23	22	21	20	19		16	15		12	11	10	9	8	7	6	5	4	3		0
1	1	1	0	0	1	1	1	1	1	0	1		Rn			Rd		0	0	0	0	0	0	0	0		Rm	

Copyright © 1996–2003 ARM Limited. All rights reserved.
Confidential - Draft

A7.1.34 LDRH (1)

15	14	13	12	11	10	6	5	3	2	0
1	0	0	0	1	immed_5		Rn		Rd	
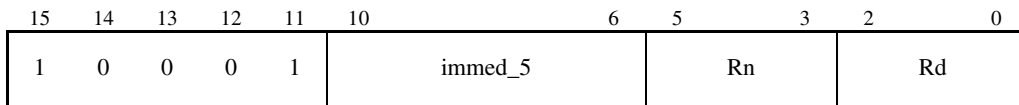
LDRH (1) (Load Register Halfword) loads a halfword (16 bits) from memory, zero-extends it to form a 32-bit word, and writes the result to a general-purpose register. The addressing mode is useful for accessing structure (record) fields. With an offset of zero, the address produced is the unaltered value of the base register <Rn>.

Syntax

```
LDRH  <Rd>, [<Rn>, #<immed_5> * 2]
```

where:

<Rd>   Is the destination register for the halfword loaded from memory.

<Rn>   Is the register containing the base address for the instruction.

<immed_5> Is a 5-bit value that is multiplied by 2, then added to the value of <Rn> to form the memory address.

Architecture version

All T variants.

Exceptions

Data Abort.

Operation

```
address = Rn + (immed_5 * 2)
if address[0] == 0
    data = Memory[address,2]
else
    data = UNPREDICTABLE
Rd = data
```

Notes

Data Abort   For details of the effects of the instruction if a Data Abort occurs, see *Effects of data-aborted instructions* on page A2-21.

Alignment   If the memory address is not halfword-aligned and no Data Abort occurs, the value written to the destination register is UNPREDICTABLE.

If an implementation includes a System Control coprocessor (see Chapter B3 *The System Control Coprocessor*) and alignment checking is enabled, an address with bit[0] != 0 causes an alignment exception (a type of Data Abort).

Equivalent ARM syntax and encoding

```
LDRH  <Rd>, [<Rn>, #<immed_5> * 2]
```

31 30 29 28	27 26 25 24 23 22 21 10	19 16	15 12	11 10	9 8	7 6 5 4	3 1	0
1 1 1 0	0 0 0 1 1 1 0 1	Rn	Rd	0 0	immed [4:3]	1 0 1 1	immed [2:0]	0

Copyright © 1996–2003 ARM Limited. All rights reserved.
Confidential - Draft

A7.1.35 LDRH (2)

15	14	13	12	11	10	9	8		6	5		3	2		0
0	1	0	1	1	0	1		Rm			Rn			Rd	

LDRH (2) loads a halfword (16 bits) from memory, zero-extends it to form a 32-bit word, and writes the result to a general-purpose register. The addressing mode is useful for pointer + large offset arithmetic and for accessing a single element of an array.

Syntax

```
LDRH  <Rd>, [<Rn>, <Rm>]
```

where:

<Rd>	Is the destination register for the halfword loaded from memory.
<Rn>	Is the register containing the first value used in forming the memory address.
<Rm>	Is the register containing the second value used in forming the memory address.

Architecture version

All T variants.

Exceptions

Data Abort.

Operation

```
address = Rn + Rm
if address[0] == 0
    data = Memory[address,2]
else
    data = UNPREDICTABLE
Rd = data
```

Notes

Data Abort For details of the effects of the instruction if a Data Abort occurs, see *Effects of data-aborted instructions* on page A2-21.

Alignment If the memory address is not halfword-aligned and no Data Abort occurs, the value written to the destination register is UNPREDICTABLE.

If an implementation includes a System Control coprocessor (see Chapter B3 *The System Control Coprocessor*) and alignment checking is enabled, an address with bit[0] != 0 causes an alignment exception (a type of Data Abort).

Equivalent ARM syntax and encoding

```
LDRH   <Rd>, [<Rn>, <Rm>]
```

31 30 29 28	27 26 25 24 23 22 21 20	19 16	15 12	11 8	7 6 5 4	3 0
1 1 1 0	0 0 0 1 1 0 0 1	Rn	Rd	SBZ	1 0 1 1	Rm

A7.1.36 LDRSB

15	14	13	12	11	10	9	8	6	5	3	2	0
0	1	0	1	0	1	1	Rm		Rn		Rd	

LDRSB (Load Register Signed Byte) loads a byte from memory, sign-extends it to form a 32-bit word, and writes the result to a general-purpose register.

Syntax

```
LDRSB   <Rd>, [<Rn>, <Rm>]
```

where:

<Rd>	Is the destination register for the byte loaded from memory.
<Rn>	Is the register containing the first value used in forming the memory address.
<Rm>	Is the register containing the second value used in forming the memory address.

Architecture version

All T variants.

Exceptions

Data Abort.

Operation

```
address = Rn + Rm
Rd = SignExtend(Memory[address,1])
```

Notes

Data Abort For details of the effects of the instruction if a Data Abort occurs, see *Effects of data-aborted instructions* on page A2-21.

Equivalent ARM syntax and encoding

```
LDRSB   <Rd>, [<Rn>, <Rm>]
```

31 30 29 28	27 26 25 24 23 22 21 20	19 16	15 12	11 8	7 6 5 4	3 0
1 1 1 0	0 0 0 1 1 0 0 1	Rn	Rd	SBZ	1 1 0 1	Rm

A7.1.37 LDRSH

15	14	13	12	11	10	9	8	6	5	3	2	0
0	1	0	1	1	1	1	Rm		Rn		Rd	

LDRSH (Load Register Signed Halfword) loads a halfword from memory, sign-extends it to form a 32-bit word, and writes the result to a general-purpose register.

Syntax

```
LDRSH  <Rd>, [<Rn>, <Rm>]
```

where:

<Rd>   Is the destination register for the halfword loaded from memory.

<Rn>   Is the register containing the first value used in forming the memory address.

<Rm>   Is the register containing the second value used in forming the memory address.

Architecture version

All T variants.

Exceptions

Data Abort.

Operation

```
address = Rn + Rm
if address[0] == 0
    data = Memory[address,2]
else
    data = UNPREDICTABLE
Rd = SignExtend(data)
```

Notes

Data Abort   For details of the effects of the instruction if a Data Abort occurs, see *Effects of data-aborted instructions* on page A2-21.

Alignment   If the memory address is not halfword-aligned and no Data Abort occurs, the value written to the destination register is UNPREDICTABLE.

If an implementation includes a System Control coprocessor (see Chapter B3 *The System Control Coprocessor*) and alignment checking is enabled, an address with bit[0] != 0 causes an alignment exception (a type of Data Abort).

Equivalent ARM syntax and encoding

```
LDRSH   <Rd>, [<Rn>, <Rm>]
```

31 30 29 28	27 26 25 24 23 22 21 20	19 16	15 12	11 8	7 6 5 4	3 0
1 1 1 0	0 0 0 1 1 0 0 1	Rn	Rd	SBZ	1 1 1 1	Rm

A7.1.38 LSL (1)

15	14	13	12	11	10	6	5	3	2	0
0	0	0	0	0	immed_5		Rm		Rd	
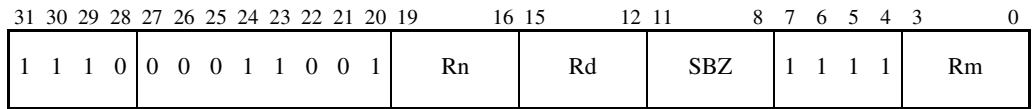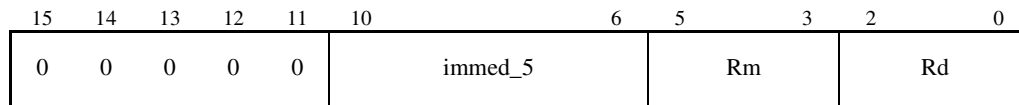
LSL (1) (Logical Shift Left) provides the value of the contents of a register multiplied by a constant power of two. It inserts zeroes into the bit positions vacated by the shift, and updates the condition code flags, based on the result.

Syntax

```
LSL  <Rd>, <Rm>, #<immed_5>
```

where:

<Rd>   Is the register that stores the result of the operation.

<Rm>   Is the register containing the value to be shifted.

<immed_5>   Specifies the shift amount, in the range 0 to 31.

Architecture version

All T variants.

Exceptions

None.

Operation

```
if immed_5 == 0
    C Flag = unaffected
    Rd = Rm
else /* immed_5 > 0 */
    C Flag = Rm[32 - immed_5]
    Rd = Rm Logical_Shift_Left immed_5
N Flag = Rd[31]
Z Flag = if Rd == 0 then 1 else 0
V Flag = unaffected
```

Usage

TBD better usage section!! You can use LSL <Rd>, <Rm>, #0 to copy a value from <Rm> to <Rd> unchanged. This has the same effect as MOV <Rd>, <Rm>.

Equivalent ARM syntax and encoding

```
MOVS   <Rd>, <Rm>, LSL #<immed_5>
```

31 30 29 28	27 26 25 24 23 22 21 20	19 16	15 12	11 7	6 5 4	3 0
1 1 1 0	0 0 0 1 1 0 1 1	SBZ	Rd	immed_5	0 0 0	Rm
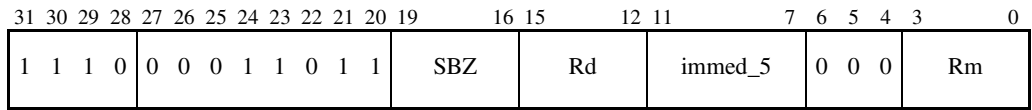
A7.1.39 LSL (2)

15	14	13	12	11	10	9	8	7	6	5		3	2		0
0	1	0	0	0	0	0	0	1	0	Rs			Rd		

LSL (2) provides the value of a register multiplied by a variable power of two. It inserts zeroes into the vacated bit positions.

It updates the condition code flags, based on the result.

Syntax

```
LSL  <Rd>, <Rs>
```

where:

<Rd>   Contains the value to be shifted, and is the destination register for the result of the operation.

<Rs>   Is the register containing the shift value. The value is held in the least significant byte.

Architecture version

All T variants.

Exceptions

None.

Operation

```
if Rs[7:0] == 0
    C Flag = unaffected
    Rd = unaffected
else if Rs[7:0] < 32 then
    C Flag = Rd[32 - Rs[7:0]]
    Rd = Rd Logical_Shift_Left Rs[7:0]
else if Rs[7:0] == 32 then
    C Flag = Rd[0]
    Rd = 0
else /* Rs[7:0] > 32 */
    C Flag = 0
    Rd = 0
N Flag = Rd[31]
Z Flag = if Rd == 0 then 1 else 0
V Flag = unaffected
```

Equivalent ARM syntax and encoding

```
MOVS   <Rd>, <Rd>, LSL <Rs>
```

31 30 29 28	27 26 25 24 23 22 21 20	19　　　　16	15　　　12	11　　　8	7 6 5 4	3　　　0
1 1 1 0	0 0 0 1 1 0 1 1	SBZ	Rd	Rs	0 0 0 1	Rd

ARM DDI 0100F-02

A7.1.40 LSR (1)

15	14	13	12	11	10	6	5	3	2	0
0	0	0	0	1	immed_5		Rm		Rd	
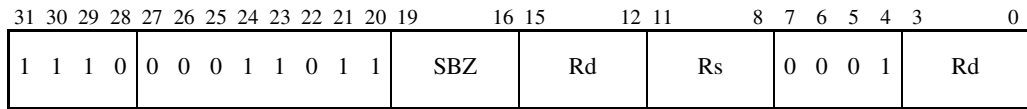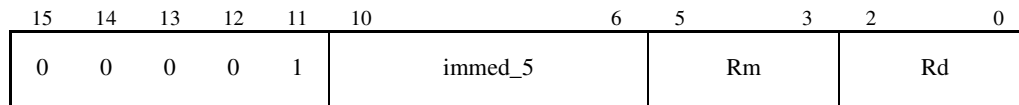
LSR (1) (Logical Shift Right) provides the unsigned value of a register, divided by a constant power of two. It inserts zeroes into the vacated bit positions.

It updates the condition code flags, based on the result.

Syntax

```
LSR  <Rd>, <Rm>, #<immed_5>
```

where:

<Rd>	Is the destination register for the operation.
<Rm>	Is the register containing the value to be shifted.
<immed_5>	Specifies the shift amount, in the range 1 to 32. Shifts by 1 to 31 are encoded directly in immed_5. A shift by 32 is encoded as immed_5 == 0.

Architecture version

All T variants.

Exceptions

None.

Operation

```
if immed_5 == 0
    C Flag = Rd[31]
    Rd = 0
else /* immed_5 > 0 */
    C Flag = Rd[immed_5 - 1]
    Rd = Rm Logical_Shift_Right immed_5
N Flag = Rd[31]
Z Flag = if Rd == 0 then 1 else 0
V Flag = unaffected
```

Equivalent ARM syntax and encoding

```
MOVS  <Rd>, <Rm>, LSR #<immed_5>
```

31 30 29 28	27 26 25 24 23 22 21 20	19 16	15 12	11 7	6 5 4	3 0
1 1 1 0	0 0 0 1 1 0 1 1	SBZ	Rd	immed_5	0 1 0	Rm
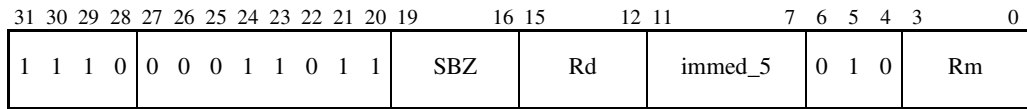
A7.1.41 LSR (2)

15	14	13	12	11	10	9	8	7	6	5		3	2		0
0	1	0	0	0	0	0	0	1	1	Rs			Rd		

LSR (2) provides the unsigned value of a register divided by a variable power of two. It inserts zeroes into the vacated bit positions.

It updates the condition code flags, based on the result.

Syntax

```
LSR  <Rd>, <Rs>
```

where:

<Rd>   Contains the value to be shifted, and is the destination register for the result of the operation.

<Rs>   Is the register containing the shift value. The value is held in the least significant byte.

Architecture version

All T variants.

Exceptions

None.

Operation

```
if Rs[7:0] == 0 then
    C Flag = unaffected
    Rd = unaffected
else if Rs[7:0] < 32 then
    C Flag = Rd[Rs[7:0] - 1]
    Rd = Rd Logical_Shift_Right Rs[7:0]
else if Rs[7:0] == 32 then
    C Flag = Rd[31]
    Rd = 0
else /* Rs[7:0] > 32 */
    C Flag = 0
    Rd = 0
N Flag = Rd[31]
Z Flag = if Rd == 0 then 1 else 0
V Flag = unaffected
```

Equivalent ARM syntax and encoding

```
MOVS  <Rd>, <Rd>, LSR <Rs>
```

31 30 29 28	27 26 25 24 23 22 21 20	19 16	15 12	11 8	7 6 5 4	3 0
1 1 1 0	0 0 0 1 1 0 1 1	SBZ	Rd	Rs	0 0 1 1	Rd

A7.1.42 MOV (1)

15	14	13	12	11	10		8	7		0
0	0	1	0	0		Rd			immed_8	
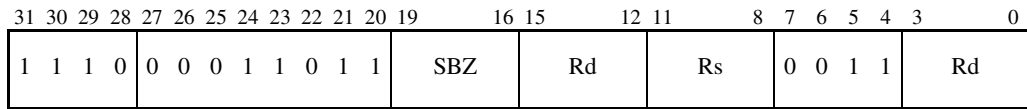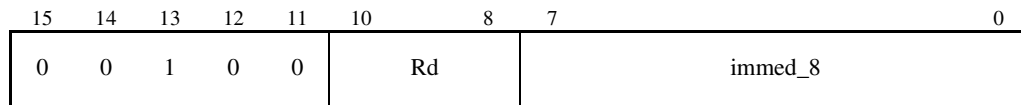
MOV (1) (Move) moves a large immediate value to a register.

It updates the condition code flags, based on the result.

Syntax

```
MOV  <Rd>, #<immed_8>
```

where:

<Rd>   Is the destination register for the operation.

<immed_8>   Is an 8-bit immediate value, in the range 0 to 255, to move into <Rd>.

Architecture version

All T variants.

Exceptions

None.

Operation

```
Rd = immed_8
N Flag = Rd[31]
Z Flag = if Rd == 0 then 1 else 0
C Flag = unaffected
V Flag = unaffected
```

Equivalent ARM syntax and encoding

```
MOVS   <Rd>, #<immed_8>
```

31 30 29 28	27 26 25 24 23 22 21 20	19 16	15 12	11 10 9 8	7 0
1 1 1 0	0 0 1 1 1 0 1 1	SBZ	Rd	0 0 0 0	immed_8

Copyright © 1996–2003 ARM Limited. All rights reserved.
Confidential - Draft

A7.1.43 MOV (2)

15	14	13	12	11	10	9	8	7	6	5		3	2		0
0	0	0	1	1	1	0	0	0	0		Rn			Rd	

MOV (2) moves a value from one low register to another.

It updates the condition code flags, based on the value.

Syntax

```
MOV  <Rd>, <Rn>
```

where:

<Rd>   Is the destination register for the operation.

<Rn>   Is the register containing the value to be copied.

Architecture Version

All T variants.

Exceptions

None.

Operation

```
Rd = Rn
N Flag = Rd[31]
Z Flag = if Rd == 0 then 1 else 0
C Flag = 0
V Flag = 0
```

Notes

Encoding   This instruction is encoded as ADD Rd, Rn, #0.

See also *ADD (1)* on page A7-5.

Confidential - Draft

Equivalent ARM syntax and encoding

```
ADDS   <Rd>, <Rn>, #0
```

31 30 29 28	27 26 25 24 23 22 21 20	19 16	15 12	11 10 9 8 7 6 5 4 3 2 1 0
1 1 1 0	0 0 1 0 1 0 0 1	Rn	Rd	0 0 0 0 0 0 0 0 0 0 0 0
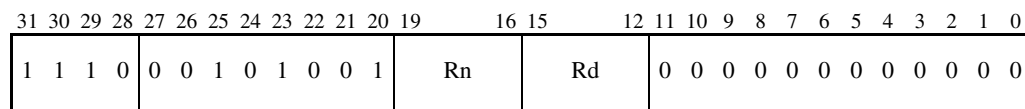
A7.1.44 MOV (3)

15	14	13	12	11	10	9	8	7	6	5		3	2		0
0	1	0	0	0	1	1	0	H1	H2	Rm			Rd		

MOV (3) moves a value to, from, or between high registers.

Unlike the low register MOV instruction described in *MOV (2)* on page A7-72, this instruction does not change the flags.

Syntax

MOV <Rd>, <Rm>

where:

<Rd>   Is the destination register for the operation. It can be any of R0 to R15, and its number is encoded in the instruction in H1 (most significant bit) and Rd (remaining three bits).

<Rm>   Is the register containing the value to be copied. It can be any of R0 to R15, and its number is encoded in the instruction in H2 (most significant bit) and Rm (remaining three bits).

Architecture version

All T variants.

Exceptions

None.

Operation

Rd = Rm

Usage

The instruction MOV PC,R14 can be used as a subroutine return instruction if it is known that the caller is also a Thumb routine. However, it is more usual to use BX R14 (see *BX* on page A7-32), which works regardless of whether the caller is an ARM routine or a Thumb routine.

In T variants of ARM architecture 4, the instruction MOV R14, PC is often used to set up a return link value in R14 for a call to an ARM routine. In T variants of ARM architecture 5 and above, there are more efficient ways of doing this. See *BX* on page A7-32 for more details.

Notes

Assembler syntax   If a low register is specified for both `<Rd>` and `<Rm>`, the assembler syntax `MOV <Rd>, <Rm>` is assembled to the `MOV` (2) instruction described on page A7-72.

Both registers low   If H1==0 and H2==0 in the encoding, the instruction specifies a non-flag-setting copy move from one low register to another low register. This instruction cannot be written using the `MOV` syntax, because `MOV <Rd>, <Rm>` generates a flag-setting copy. However, you can write it using the `CPY` mnemonic, see *CPY* on page A7-41.

---- **Note** ----

This was not available in architecture v5 and below.

Equivalent ARM syntax and encoding

A close equivalent is:

```
MOV  <Rd>, <Rm>
```

31 30 29 28	27 26 25 24 23 22 21 20	19 16	15	14 12	11 10 9 8 7 6 5 4	3	2 0
1 1 1 0	0 0 0 1 1 0 1 0	SBZ	H1	Rd	0 0 0 0 0 0 0 0	H2	Rm

There are slight differences when the instruction accesses the PC, because of the different definitions of the PC when executing ARM and THUMB code.

A7.1.45 MUL

15	14	13	12	11	10	9	8	7	6	5		3	2		0
0	1	0	0	0	0	1	1	0	1		Rm			Rd	

MUL (Multiply) multiplies signed or unsigned variables to produce a 32-bit result.

MUL updates the condition code flags, based on the result.

Syntax

```
MUL  <Rd>, <Rm>
```

where:

<Rd>   Contains the value to be multiplied with the value of <Rm>, and is also the destination
                register for the operation.

<Rm>   Is the register containing the value to be multiplied with the value of <Rd>.

Architecture version

All T variants.

Exceptions

None.

Operation

```
Rd = (Rm * Rd)[31:0]
N Flag = Rd[31]
Z Flag = if Rd == 0 then 1 else 0
C Flag = unaffected    /* See "C flag" note */
V Flag = unaffected
```

ARM DDI 0100F-02

Notes

Early termination   If the multiplier implementation supports early termination, it must be implemented on the value of the `<Rd>` operand. The type of early termination used (signed or unsigned) is IMPLEMENTATION DEFINED.

Signed and unsigned   As `MUL` produces only the lower 32 bits of the 64-bit product, `MUL` gives the same answer for multiplication of both signed and unsigned numbers.

C flag   The `MUL` instruction is defined to leave the C flag unchanged in ARM architecture version 5 and above. In earlier versions of the architecture, the value of the C flag was UNPREDICTABLE after a `MUL` instruction.

Operand restrictions   There are no operand restrictions in ARM architecture version 6 and above. In earlier versions of the architecture, specifying the same register for `<Rd>` and `<Rm>` had UNPREDICTABLE results.

Equivalent ARM syntax and encoding

```
MULS    <Rd>, <Rm>, <Rd>
```

31 30 29 28	27 26 25 24 23 22 21 20	19 16	15 12	11 8	7 6 5 4	3 0
1 1 1 0	0 0 0 0 0 0 0 1	Rd	SBZ	Rd	1 0 0 1	Rm

─── **Note** ───

The following instruction is not a suitable alternative, as it violates the operand restriction on the ARM instruction (see *MUL* on page A4-79) and might have the wrong early termination behavior:

```
MULS    <Rd>, <Rd>, <Rm>
```

A7.1.46 MVN

15	14	13	12	11	10	9	8	7	6	5		3	2		0
0	1	0	0	0	0	1	1	1	1		Rm			Rd	

MVN (Move NOT) complements a register value. This is often used to form a bit mask.

MVN updates the condition code flags, based on the result.

Syntax

```
MVN  <Rd>, <Rm>
```

where:

<Rd>   Is the destination register for the operation.

<Rm>   Is the register containing the value whose one's complement is written to <Rd>.

Architecture version

All T variants.

Exceptions

None.

Operation

```
Rd = NOT Rm
N Flag = Rd[31]
Z Flag = if Rd == 0 then 1 else 0
C Flag = unaffected
V Flag = unaffected
```

Equivalent ARM syntax and encoding

```
MVNS    <Rd>, <Rm>
```

31	30	29	28	27	26	25	24	23	22	21	20	19		16	15		12	11	10	9	8	7	6	5	4	3		0
1	1	1	0	0	0	0	1	1	1	1	1		SBZ			Rd		0	0	0	0	0	0	0	0		Rm	

A7.1.47 NEG

15	14	13	12	11	10	9	8	7	6	5		3	2		0
0	1	0	0	0	0	1	0	0	1		Rm			Rd	

NEG (Negate) negates the value of one register and stores the result in a second register.

NEG updates the condition code flags, based on the result.

Syntax

```
NEG  <Rd>, <Rm>
```

where:

<Rd>   Is the destination register for the operation.

<Rm>   Is the register containing the value that is subtracted from zero.

Architecture version

All T variants.

Exceptions

None.

Operation

```
Rd = 0 - Rm
N Flag = Rd[31]
Z Flag = if Rd == 0 then 1 else 0
C Flag = NOT BorrowFrom(0 - Rm)
V Flag = OverflowFrom(0 - Rm)
```

Equivalent ARM syntax and encoding

```
RSBS  <Rd>, <Rm>, #0
```

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

Copyright © 1996–2003 ARM Limited. All rights reserved.
Confidential - Draft

A7.1.48 ORR

15	14	13	12	11	10	9	8	7	6	5		3	2		0
0	1	0	0	0	0	1	1	0	0		Rm			Rd	

ORR (Logical OR) performs a bitwise OR of the values from two registers.

ORR updates the condition code flags, based on the result.

Syntax

```
ORR  <Rd>, <Rm>
```

where:

<Rd>   Is the destination register for the operation.

<Rm>   Is the register containing the value that is ORed with the value of <Rd>. The operation is a bitwise inclusive OR.

Architecture version

All T variants.

Exceptions

None.

Operation

```
Rd = Rd OR Rm
N Flag = Rd[31]
Z Flag = if Rd == 0 then 1 else 0
C Flag = unaffected
V Flag = unaffected
```

Operation

```
ORRS  <Rd>, <Rd>, <Rm>
```

31	30	29	28	27	26	25	24	23	22	21	20	19		16	15		12	11	10	9	8	7	6	5	4	3		0
1	1	1	0	0	0	0	1	1	0	0	1		Rd			Rd		0	0	0	0	0	0	0	0		Rm	

A7.1.49 POP

15	14	13	12	11	10	9	8	7	0
1	0	1	1	1	1	0	R	register_list	
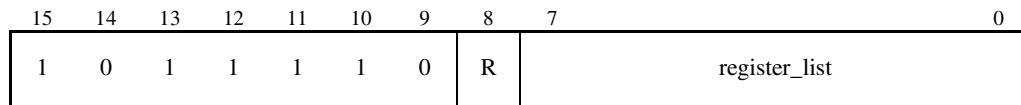
POP (Pop Multiple Registers) loads a subset (or possibly all) of the general-purpose registers R0-R7 and the PC from the stack.

The general-purpose registers loaded can include the PC. If they do, the word loaded for the PC is treated as an address and a branch occurs to that address. In ARM architecture version 5 and above, bit[0] of the loaded value determines whether execution continues after this branch in ARM state or in Thumb state, as though the following instruction had been executed:

```
BX (loaded_value)
```

In T variants of architecture version 4, bit[0] of the loaded value is ignored and execution continues in Thumb state, as though the following instruction had been executed:

```
MOV PC,(loaded_value)
```

Syntax

```
POP  <registers>
```

where:

<registers> Is the list of registers, separated by commas and surrounded by { and }. The list is encoded in the register_list field of the instruction, by setting bit[i] to 1 if register Ri is included in the list and to 0 otherwise, for each of i=0 to 7. The R bit (bit[8]) is set to 1 if the PC is in the list and to 0 otherwise.

At least one register must be loaded. If bits[8:0] are all zero, the result is UNPREDICTABLE.

The registers are loaded in sequence, the lowest-numbered register from the lowest memory address (start_address), through to the highest-numbered register from the highest memory address (end_address). If the PC is specified in the register list (opcode bit[8] is set), the instruction causes a branch to the address (data) loaded into the PC.

The <start_address> is the value of the SP.

Subsequent addresses are formed by incrementing the previous address by four. One address is produced for each register that is specified in <registers>.

The end_address value is four less than the sum of the value of the SP and four times the number of registers specified in <registers>.

The SP register is incremented by four times the numbers of registers in <registers>.

Architecture version

All T variants.

Exceptions

Data Abort.

Operation

```
start_address = SP
end_address = SP + 4*(R + Number_Of_Set_Bits_In(register_list))
address = start_address

for i = 0 to 7
    if register_list[i] == 1 then
        Ri = Memory[address,4]
        address = address + 4

if R == 1 then
    value = Memory[address,4]
    PC = value AND 0xFFFFFFFE
    if (architecture version 5 or above) then
        T Bit = value[0]
    address = address + 4

assert end_address = address
SP = end_address
```

Usage

Use POP for stack operations. A POP instruction with the PC in the register list can be used for an efficient procedure exit, as it restores saved registers, loads the PC with the return address, and updates the stack pointer with a single instruction.

Notes

Data Abort   For details of the effects of the instruction if a Data Abort occurs, see *Effects of data-aborted instructions* on page A2-21.

CPSR   The CPSR is not updated when POP loads the PC.

Alignment   POP instructions ignore the least significant two bits of address.

If an implementation includes a System Control coprocessor (see Chapter B3 *The System Control Coprocessor*) and alignment checking is enabled, an address with bits[1:0] != 0b00 causes an alignment exception.

ARM DDI 0100F-02

ARM/Thumb state transfers

In ARM architecture 5 and above, if bits[1:0] of a value loaded for R15 are 0b10, the result is UNPREDICTABLE, as branches to non word-aligned addresses are not possible in ARM state.

Time order   The time order of the accesses to individual words of memory generated by this instruction is only defined in some circumstances. See *Data accesses to memory-mapped I/O* on page B2-27 for details.

Equivalent ARM syntax and encoding

```
LDMIA  SP!, <registers>
```

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 0
1 1 1 0	1 0 0 0 1 0 1 1	1 1 0 1	R	0 0 0 0 0 0 0	register_list

A7.1.50 PUSH

15	14	13	12	11	10	9	8	7	0
1	0	1	1	0	1	0	R	register_list	
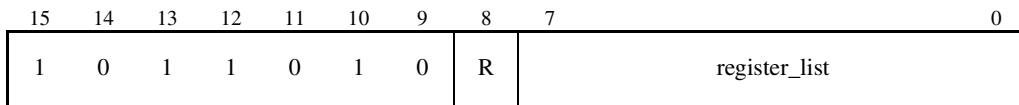
PUSH (Push Multiple Registers) stores a subset (or possibly all) of the general-purpose registers R0-R7 and the LR to the stack.

Syntax

```
PUSH   <registers>
```

where:

<registers>   Is the list of registers to be stored, separated by commas and surrounded by { and }. The list is encoded in the register_list field of the instruction, by setting bit[i] to 1 if register Ri is included in the list and to 0 otherwise, for each of i=0 to 7. The R bit (bit[8]) is set to 1 if the LR is in the list and to 0 otherwise.

At least one register must be stored. If bits[8:0] are all zero, the result is UNPREDICTABLE.

The registers are stored in sequence, the lowest-numbered register to the lowest memory address (start_address), through to the highest-numbered register to the highest memory address (end_address)

The start_address is the value of the SP minus 4 times the number of registers to be stored.

Subsequent addresses are formed by incrementing the previous address by four. One address is produced for each register that is specified in <registers>.

The end_address value is four less than the original value of SP.

The SP register is decremented by four times the numbers of registers in <registers>.

Architecture version

All T variants.

Exceptions

Data Abort.

Operation

```
start_address = SP - 4*(R + Number_Of_Set_Bits_In(register_list))
end_address = SP - 4
address = start_address
for i = 0 to 7
    if register_list[i] == 1
        Memory[address,4] = Ri
        address = address + 4
if R == 1
    Memory[address,4] = LR
    address = address + 4
assert end_address == address - 4
SP = SP - 4*(R + Number_Of_Set_Bits_In(register_list))
```

Usage

Use PUSH for stack operations. A PUSH instruction with the LR in the register list can be used for an efficient procedure entry, as it saves registers (including the return address) on the stack and updates the stack pointer with a single instruction. A matching POP instruction can be used later to return from the procedure.

Notes

Data Abort   For details of the effects of the instruction if a Data Abort occurs, see *Effects of data-aborted instructions* on page A2-21.

Alignment   PUSH instructions ignore the least significant two bits of address.

If an implementation includes a System Control coprocessor (see Chapter B3 *The System Control Coprocessor*) and alignment checking is enabled, an address with bits[1:0] != 0b00 causes an alignment exception.

Time order   The time order of the accesses to individual words of memory generated by this instruction is only defined in some circumstances. See *Data accesses to memory-mapped I/O* on page B2-27 for details.

Equivalent ARM syntax and encoding

```
STMDB  SP!, <registers>
```

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 0
1 1 1 0	1 0 0 1 0 0 1 0	1 1 0 1	0	R	0 0 0 0 0 0	register_list

Confidential - Draft

A7.1.51 REV

15			12	11			8	7	6	5		3	2		0
1	0	1	1	1	0	1	0	0	0		Rn			Rd	
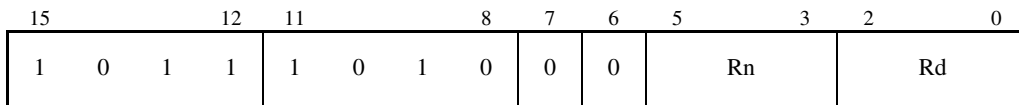
REV (Byte-Reverse Word) reverses the byte order in a 32-bit register.

Syntax

```
REV Rd, Rm
```

where:

<Rd>   Specifies the destination register.

<Rn>   Specifies the register that contains the operand.

Architecture version

Version 6 and above.

Exceptions

None.

Operation

```
Rd[31:24] = Rm[ 7: 0]
Rd[23:16] = Rm[15: 8]
Rd[15: 8] = Rm[23:16]
Rd[ 7: 0] = Rm[31:24]
```

Usage

Use REV to convert 32-bit big-endian data into little-endian data, or 32-bit little-endian data into big-endian data.

Equivalent ARM syntax and encoding

```
REV Rd, Rm
```
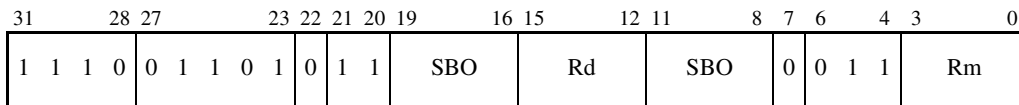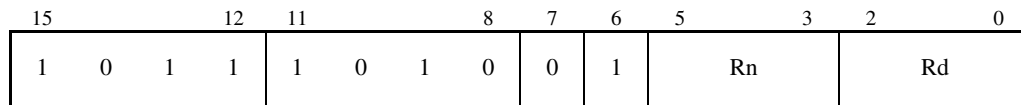
31			28	27				23	22	21	20	19		16	15		12	11		8	7	6		4	3		0
1	1	1	0	0	1	1	0	1	0	1	1		SBO			Rd			SBO		0	0	1	1		Rm	

A7.1.52 REV16

15			12	11			8	7	6	5		3	2		0
1	0	1	1	1	0	1	0	0	1		Rn			Rd	

`REV16` (Byte-Reverse Packed Halfword) reverses the byte order in each 16-bit halfword of a 32-bit register.

Syntax

`REV16 Rd, Rm`

where:

`<Rd>`   Specifies the destination register.

`<Rn>`   Specifies the register that contains the operand.

Architecture version

Version 6 and above.

Exceptions

None.

Operation

```
Rd[15: 8] = Rm[ 7: 0]
Rd[ 7: 0] = Rm[15: 8]
Rd[31:24] = Rm[23:16]
Rd[23:16] = Rm[31:24]
```

Usage

Use `REV16` to convert 16-bit big-endian data into little-endian data, or 16-bit little-endian data into big-endian data.

Equivalent ARM syntax and encoding

`REV16 Rd, Rm`

31			28	27				23	22	21	20	19		16	15		12	11		8	7	6		4	3		0
1	1	1	0	0	1	1	0	1	0	1	1		SBO			Rd			SBO		1	0	1	1		Rm	

Copyright © 1996–2003 ARM Limited. All rights reserved.
Confidential - Draft

A7.1.53 REVSH

15			12	11			8	7	6	5		3	2		0
1	0	1	1	1	0	1	0	1	1		Rn			Rd	
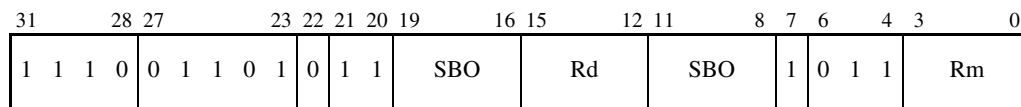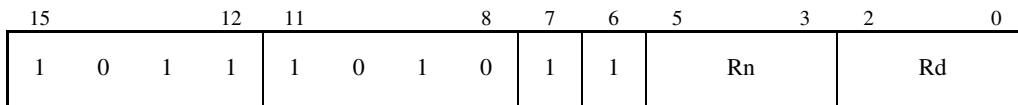
REV16 (Byte-Reverse Signed Halfword) reverses the byte order in the lower 16-bit halfword of a 32-bit register, and sign extends the result to 32-bits.

Syntax

```
REV16 Rd, Rm
```

where:

<Rd>   Specifies the destination register.

<Rn>   Specifies the register that contains the operand.

Architecture version

Version 6 and above.

Exceptions

None.

Operation

```
Rd[15: 8] = Rm[ 7: 0]
Rd[ 7: 0] = Rm[15: 8]
if Rm[7] == 1 then
    Rd[31:16] = 0xFFFF
else                          TBD Please check this carefully - it's not
    Rd[31:16] = 0x0000            what ARCH v6 SPEC 0002 E says!
```

Usage

Use REVSH to convert either:
- 16-bit signed big-endian data into 32-bit signed little-endian data
- 16-bit signed little-endian data into 32-bit signed big-endian data.

Equivalent ARM syntax and encoding

```
REVSH Rd, Rm
```

31			28	27				23	22	21	20	19		16	15		12	11		8	7	6		4	3		0
1	1	1	0	0	1	1	0	1	1	1	1	SBO			Rd			SBO			1	0	1	1	Rm		
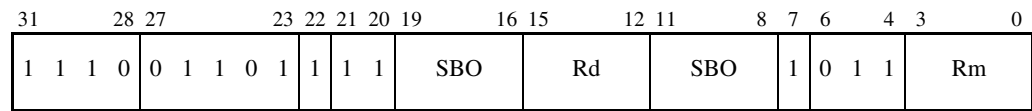
A7.1.54 ROR

15	14	13	12	11	10	9	8	7	6	5		3	2		0
0	1	0	0	0	0	0	1	1	1		Rs			Rd	

ROR (Rotate Right Register) provides the value of the contents of a register rotated by a variable value. The bits that are rotated off the right end are inserted into the vacated bit positions on the left.

ROR updates the condition code flags, based on the result.

Syntax

```
ROR  <Rd>, <Rs>
```

where:

<Rd>   Contains the value to be rotated, and is also the destination register for the operation.

<Rs>   Is the register containing the rotation applied to the value of <Rd>. The value of the rotation is stored in the least significant byte.

Architecture version

All T variants.

Exceptions

None.

Operation

```
if Rs[7:0] == 0 then
    C Flag = unaffected
    Rd = unaffected
else if Rs[4:0] == 0 then
    C Flag = Rd[31]
    Rd = unaffected
else /* Rs[4:0] > 0 */
    C Flag = Rd[Rs[4:0] - 1]
    Rd = Rd Rotate_Right Rs[4:0]
N Flag = Rd[31]
Z Flag = if Rd == 0 then 1 else 0
V Flag = unaffected
```

Equivalent ARM syntax and encoding

```
MOVS   <Rd>, <Rd>, ROR <Rs>
```

31 30 29 28	27 26 25 24 23 22 21 20	19 16	15 12	11 8	7 6 5 4	3 0
1 1 1 0	0 0 0 1 1 0 1 1	SBZ	Rd	Rs	0 1 1 1	Rd
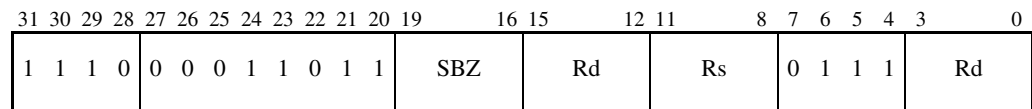
A7.1.55 SBC

15	14	13	12	11	10	9	8	7	6	5		3	2		0
0	1	0	0	0	0	0	1	1	0	Rm			Rd		

SBC (Subtract with Carry) subtracts the value of its second operand and the value of NOT(Carry flag) from the value of its first operand.

SBC updates the condition code flags, based on the result.

Use SBC to synthesize multi-word subtraction.

Syntax

```
SBC   <Rd>, <Rm>
```

where:

<Rd>   Contains the first operand for the subtraction, and is also the destination register for the operation.

<Rm>   Contains the value to be subtracted from <Rd>.

Architecture version

All T variants.

Exceptions

None.

Operation

```
Rd = Rd - Rm - NOT(C Flag)
N Flag = Rd[31]
Z Flag = if Rd == 0 then 1 else 0
C Flag = NOT BorrowFrom(Rd - Rm - NOT(C Flag))
V Flag = OverflowFrom(Rd - Rm - NOT(C Flag))
```

Equivalent ARM syntax and encoding

```
SBCS   <Rd>, <Rd>, <Rm>
```

31 30 29 28	27 26 25 24 23 22 21 20	19 16	15 12	11 10 9 8 7 6 5 4	3 0
1 1 1 0	0 0 0 0 1 1 0 1	Rd	Rd	0 0 0 0 0 0 0 0	Rm

A7.1.56 SETEND

15	14	13	12	11	10	9	8	7	6	5		3	2		0
1	0	1	1	0	1	1	0	0	1	0	1	E		SBZ	

SETEND modifies the CPSR E bit, without changing any other bits in the CPSR.

Syntax

```
SETEND <endian_specifier>
```

where:

```
<endian_specifier>
```

> Is one of:
>
> BE   Sets the E bit in the instruction. This sets the CPSR E bit.
>
> LE   Clears the E bit in the instruction. This clears the CPSR E bit.

Architecture version

Version 6 and above.

Exceptions

None.

Operation

```
CPSR = CPSR with specified E bit modification
```

Usage

Use SETEND to change the byte order for data accesses. You can use SETEND to increase the efficiency of access to a series of big-endian data fields in an otherwise little-endian application, or to a series of little-endian data fields in an otherwise big-endian application.

Equivalent ARM syntax and encoding

```
SETEND <endian_specifier>
```

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15		10	9	8	7			4	3			0
1	1	1	1	0	0	0	1	0	0	0	0	0	0	0	1	SBZ			E	SBZ	0	0	0	0	SBZ			

Copyright © 1996–2003 ARM Limited. All rights reserved.
Confidential - Draft

A7.1.57 SEXT16

15	14	13	12	11	10	9	8	7	6	5		3	2		0
1	0	1	1	0	0	1	0	0	0		Rm			Rd	

SEXT16 (Sign Extend 16) extracts the least significant 16 bits of the operand, and sign extends the value to 32 bits.

SEXT16 does not affect the flags.

Syntax

SEXT16   <Rd>, <Rm>

where:

<Rd>   Specifies the destination register.

<Rm>   Specifies the operand register.

Architecture version

Version 6 and above.

Exceptions

None.

Operation

Rd = SignExtend(Rm[15:0])

Usage

Use SEXT16 to sign extend a halfword to a word, for example in instruction sequences acting on **signed short** values in C/C++.

Equivalent ARM syntax and encoding

SUNPK16TO32   <Rd>, <Rm>

31		28	27	26	25	24	23	22	21	20	19	18	17	16	15		12	11	10	9	8	7	6	5	4	3		0
	Cond		0	1	1	0	1	0	1	1	1	1	1	1		Rd		0	0	0	0	0	1	1	1		Rm	

A7.1.58 SEXT8

15	14	13	12	11	10	9	8	7	6	5		3	2		0
1	0	1	1	0	0	1	0	0	1	Rm			Rd		

SEXT8 (Sign Extend 8) extracts the least significant 8 bits of the operand, and sign extends the value to 32 bits.

SEXT8 does not affect the flags.

Syntax

SEXT8   <Rd>, <Rm>

where:

<Rd>   Specifies the destination register.

<Rm>   Specifies the operand register.

Architecture version

Version 6 and above.

Exceptions

None.

Operation

Rd = SignExtend(Rm[7:0])

Usage

Use SEXT8 to sign extend a byte to a word, for example in instruction sequences acting on **signed char** values in C/C++.

Equivalent ARM syntax and encoding

SUNPK8TO32   <Rd>, <Rm>

31	28	27	26	25	24	23	22	21	20	19	18	17	16	15		12	11	10	9	8	7	6	5	4	3		0
Cond		0	1	1	0	1	0	1	0	1	1	1	1	Rd			0	0	0	0	0	1	1	1	Rm		

Copyright © 1996–2003 ARM Limited. All rights reserved.
Confidential - Draft

A7.1.59 STMIA

15	14	13	12	11	10	8	7	0
1	1	0	0	0		Rn	register_list	
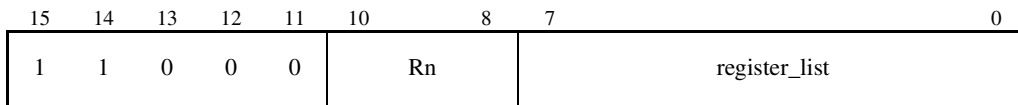
STMIA (Store Multiple Increment After) stores a non-empty subset, or possibly all, of the general-purpose registers to sequential memory locations.

Syntax

STMIA   <Rn>!, <registers>

where:

<Rn>	Is the register containing the start address for the instruction.
!	Causes base register write-back, and is not optional.
<registers>	Is a list of registers to be stored, separated by commas and surrounded by { and }. The list is encoded in the register_list field of the instruction, by setting bit[i] to 1 if register Ri is included in the list and to 0 otherwise, for each of i=0 to 7.
	At least one register must be stored. If bits[7:0] are all zero, the result is UNPREDICTABLE.
	The registers are stored in sequence, the lowest-numbered register to the lowest memory address (start_address), through to the highest-numbered register to the highest memory address (end_address).
	The start_address is the value of the base register <Rn>. Subsequent addresses are formed by incrementing the previous address by four. One address is produced for each register that is specified in <registers>.
	The end_address value is four less than the sum of the value of the base register and four times the number of registers specified in <registers>.
	Finally, the base register <Rn> is incremented by 4 times the numbers of registers in <registers>.

Architecture version

All T variants.

Exceptions

Data Abort.

Operation

```
start_address = Rn
end_address = Rn + (Number_Of_Set_Bits_In(register_list) * 4) - 4
address = start_address
for i = 0 to 7
    if register_list[i] == 1
        Memory[address,4] = Ri
        address = address + 4
assert end_address == address - 4
Rn = Rn + (Number_Of_Set_Bits_In(register_list) * 4)
```

Usage

STMIA is useful as a block store instruction. Combined with LDMIA (Load Multiple), it allows efficient block copy.

Notes

Operand restrictions

If <Rn> is specified in <registers> and write-back is specified:

- If <Rn> is the lowest-numbered register specified in <registers>, the original value of <Rn> is stored.

- Otherwise, the stored value of <Rn> is UNPREDICTABLE.

Data Abort   For details of the effects of the instruction if a Data Abort occurs, see *Effects of data-aborted instructions* on page A2-21.

Alignment   Store Multiple instructions ignore the least significant two bits of address.

If an implementation includes a System Control coprocessor (see Chapter B3 *The System Control Coprocessor*) and alignment checking is enabled, an address with bits[1:0] != 0b00 causes an alignment exception.

Time order   The time order of the accesses to individual words of memory generated by this instruction is only defined in some circumstances. See *Data accesses to memory-mapped I/O* on page B2-27 for details.

Equivalent ARM syntax and encoding

```
STMIA   <Rn>!, <registers>
```

31 30 29 28	27 26 25 24 23 22 21 20	19 16	15 14 13 12 11 10 9 8	7 0
1 1 1 0	1 0 0 0 1 0 1 0	Rn	0 0 0 0 0 0 0 0	register_list

Copyright © 1996–2003 ARM Limited. All rights reserved.
Confidential - Draft

A7.1.60 STR (1)

15	14	13	12	11	10			6	5		3	2		0
0	1	1	0	0		immed_5				Rn			Rd	
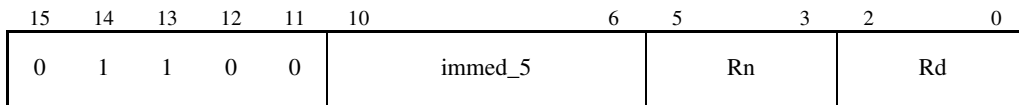
STR (1) (Store Register) stores 32-bit data from a general-purpose register to memory. The addressing mode is useful for accessing structure (record) fields. With an offset of zero, the address produced is the unaltered value of the base register <Rn>.

Syntax

```
STR  <Rd>, [<Rn>, #<immed_5> * 4]
```

where:

<Rd>   Is the register that contains the word to be stored to memory.

<Rn>   Is the register containing the base address for the instruction.

<immed_5> Is a 5-bit value that is multiplied by 4 and added to the value of <Rn> to form the memory address.

Architecture version

All T variants.

Exceptions

Data Abort.

Operation

```
address = Rn + (immed_5 * 4)
if address[1:0] == 0b00
    Memory[address,4] = Rd
else
    Memory[address,4] = UNPREDICTABLE
```

Notes

Data Abort   For details of the effects of the instruction if a Data Abort occurs, see *Effects of data-aborted instructions* on page A2-21.

Alignment   If the memory address is not word-aligned and no Data Abort occurs, the value written to memory is UNPREDICTABLE.

If an implementation includes a System Control coprocessor (see Chapter B3 *The System Control Coprocessor*) and alignment checking is enabled, an address with bits[1:0] != 0b00 causes an alignment exception (a type of Data Abort).

Equivalent ARM syntax and encoding

```
STR  <Rd>, [<Rn>, #<immed_5> * 4]
```

31 30 29 28	27 26 25 24 23 22 21 20	19 16	15 12	11 10 9 8	7 6 2	1 0
1 1 1 0	0 1 0 1 1 0 0 0	Rn	Rd	0 0 0 0	immed_5	0 0

A7.1.61 STR (2)

15	14	13	12	11	10	9	8	6	5	3	2	0
0	1	0	1	0	0	0	Rm		Rn		Rd	

STR (2) stores 32-bit data from a general-purpose register to memory. The addressing mode is useful for pointer + large offset arithmetic, and for accessing a single element of an array.

Syntax

```
STR  <Rd>, [<Rn>, <Rm>]
```

where:

<Rd>   Is the register that contains the word to be stored to memory.

<Rn>   Is the register containing the first value used in forming the memory address.

<Rm>   Is the register containing the second value used in forming the memory address.

Architecture version

All T variants.

Exceptions

Data Abort.

Operation

```
address = Rn + Rm
if address[1:0] == 0b00
    Memory[address,4] = Rd
else
    Memory[address,4] = UNPREDICTABLE
```

Notes

Data Abort   For details of the effects of the instruction if a Data Abort occurs, see *Effects of data-aborted instructions* on page A2-21.

Alignment   If the memory address is not word-aligned and no Data Abort occurs, the value written to memory is UNPREDICTABLE.

If an implementation includes a System Control coprocessor (see Chapter B3 *The System Control Coprocessor*) and alignment checking is enabled, an address with bits[1:0] != 0b00 causes an alignment exception (a type of Data Abort).

Equivalent ARM syntax and encoding

```
STR  <Rd>, [<Rn>, <Rm>]
```

31 30 29 28	27 26 25 24 23 22 21 20	19 16	15 12	11 10 9 8 7 6 5 4	3 0
1 1 1 0	0 1 1 1 1 0 0 0	Rn	Rd	0 0 0 0 0 0 0 0	Rm

A7.1.62 STR (3)

15	14	13	12	11	10 8	7 0
1	0	0	1	0	Rd	immed_8

STR (3) stores 32-bit data from a general-purpose register to memory. The addressing mode is useful for accessing stack data. In this case, STR stores a word from register <Rd> to memory.

Syntax

```
STR  <Rd>, [SP, #<immed_8> * 4]
```

where:

<Rd>   Is the register that contains the word to be stored to memory.

SP   Is the stack pointer. Its value is used to calculate the memory address.

<immed_8>   Is an 8-bit value that is multiplied by 4 and added to the value of the SP to form the memory address.

Architecture version

All T variants.

Exceptions

Data Abort.

Operation

```
address = SP + (immed_8 * 4)
if address[1:0] == 0b00
    Memory[address,4] = Rd
else
    Memory[address,4] = UNPREDICTABLE
```

Notes

Data Abort   For details of the effects of the instruction if a Data Abort occurs, see *Effects of data-aborted instructions* on page A2-21.

Alignment   If the memory address is not word-aligned and no Data Abort occurs, the value written to memory is UNPREDICTABLE.

If an implementation includes a System Control coprocessor (see Chapter B3 *The System Control Coprocessor*), and alignment checking is enabled, an address with bits[1:0] != 0b00 causes an alignment exception (a type of Data Abort).

Equivalent ARM syntax and encoding

```
STR  <Rd>, [SP, #<immed_8> * 4]
```

31 30 29 28	27 26 25 24 23 22 21 20	19 18 17 16	15 12	11 10	9 2	1 0
1 1 1 0	0 1 0 1 1 0 0 0	1 1 0 1	Rd	0 0	immed_8	0 0

A7.1.63 STRB (1)

15	14	13	12	11	10	6	5	3	2	0
0	1	1	1	0	immed_5		Rn		Rd	
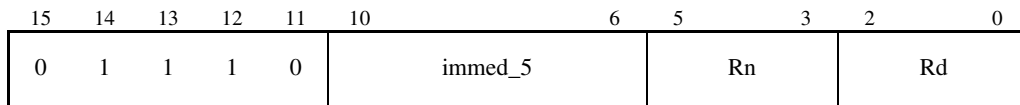
STRB (1) (Store Register Byte) stores 8-bit data from a general-purpose register to memory. The addressing mode is useful for accessing structure (record) fields.

With an offset of zero, the address produced is the unaltered value of the base register <Rn>.

Syntax

```
STRB   <Rd>, [<Rn>, #<immed_5>]
```

where:

<Rd>   Is the register whose least significant byte is stored to memory.

<Rn>   Is the register containing the base address for the instruction.

<immed_5>   Is a 5-bit immediate value that is added to the value of <Rn> to form the memory address.

Architecture version

All T variants.

Exceptions

Data Abort.

Operation

```
address = Rn + immed_5
Memory[address,1] = Rd[7:0]
```

Notes

Data Abort   For details of the effects of the instruction if a Data Abort occurs, see *Effects of data-aborted instructions* on page A2-21.

Copyright © 1996–2003 ARM Limited. All rights reserved.
Confidential - Draft

Equivalent ARM syntax and encoding

```
STRB   <Rd>, [<Rn>, #<immed_5>]
```

31 30 29 28	27 26 25 24 23 22 21 20	19 16	15 12	11 10 9 8 7 6 5	4 0
1 1 1 0	0 1 0 1 1 1 0 0	Rn	Rd	0 0 0 0 0 0 0	immed_5
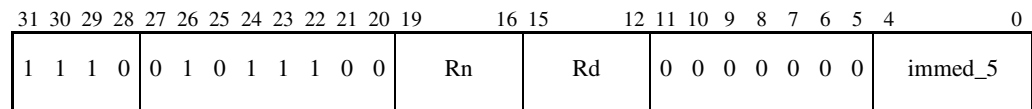
A7.1.64 STRB (2)

15	14	13	12	11	10	9	8		6	5		3	2		0
0	1	0	1	0	1	0		Rm			Rn			Rd	

STRB (2) stores 8-bit data from a general-purpose register to memory. The addressing mode is useful for pointer + large offset arithmetic, and for accessing a single element of an array.

Syntax

```
STRB  <Rd>, [<Rn>, <Rm>]
```

where:

<Rd>   Is the register whose least significant byte is stored to memory.

<Rn>   Is the register containing the first value used in forming the memory address.

<Rm>   Is the register whose value is added to <Rn> to form the memory address.

Architecture version

All T variants.

Exceptions

Data Abort.

Operation

```
address = Rn + Rm
Memory[address,1] = Rd[7:0]
```

Notes

Data Abort   For details of the effects of the instruction if a Data Abort occurs, see *Effects of data-aborted instructions* on page A2-21.

Equivalent ARM syntax and encoding

```
STRB  <Rd>, [<Rn>, <Rm>]
```

31 30 29 28	27 26 25 24 23 22 21 20	19 16	15 12	11 10 9 8 7 6 5 4	3 0
1 1 1 0	0 1 1 1 1 1 0 0	Rn	Rd	0 0 0 0 0 0 0 0	Rm

ARM DDI 0100F-02

A7.1.65 STRH (1)

15	14	13	12	11	10	6	5	3	2	0
1	0	0	0	0	immed_5		Rn		Rd	
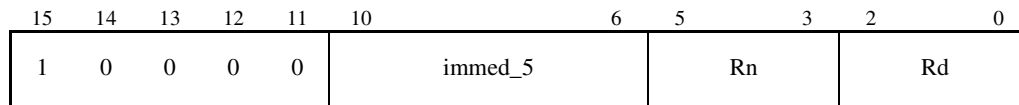
STRH (1) (Store Register Halfword) stores 16-bit data from a general-purpose register to memory. The addressing mode is useful for accessing structure (record) fields. With an offset of zero, the address produced is the unaltered value of the base register <Rn>.

Syntax

```
STRH  <Rd>, [<Rn>, #<immed_5> * 2]
```

where:

<Rd>   Is the register whose least significant halfword is stored to memory.

<Rn>   Is the register containing the base address for the instruction.

<immed_5>   Is a 5-bit immediate value that is multiplied by two and added to the value of <Rn> to form
            the memory address.

Architecture version

All T variants.

Exceptions

Data Abort.

Operation

```
address = Rn + (immed_5 * 2)
if address[1:0] == 0
    Memory[address,2] = Rd[15:0]
else
    Memory[address,2] = UNPREDICTABLE
```

Notes

Data Abort   For details of the effects of the instruction if a Data Abort occurs, see *Effects of data-aborted instructions* on page A2-21.

Alignment   If the memory address is not halfword-aligned and no Data Abort occurs, the value written to memory is UNPREDICTABLE.

If an implementation includes a System Control coprocessor (see Chapter B3 *The System Control Coprocessor*) and alignment checking is enabled, an address with bit[0] != 0 causes an alignment exception (a type of Data Abort).

Equivalent ARM syntax and encoding

```
STRH  <Rd>, [<Rn>, #<immed_5> * 2]
```

31 30 29 28	27 26 25 24 23 22 21 20	19　　　　16	15　　　　12	11 10	9　8	7 6 5 4	3　　　1	0
1 1 1 0	0 0 0 1 1 1 0 0	Rn	Rd	0 0	immed [4:3]	1 0 1 1	immed [2:0]	0

A7.1.66 STRH (2)

15	14	13	12	11	10	9	8		6	5		3	2		0
0	1	0	1	0	0	1		Rm			Rn			Rd	
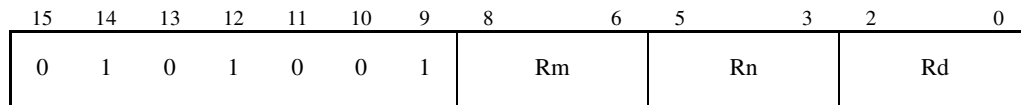
STRH (2) stores 16-bit data from a general-purpose register to memory. The addressing mode is useful for pointer + large offset arithmetic and for accessing a single element of an array.

Syntax

```
STRH  <Rd>, [<Rn>, <Rm>]
```

where:

<Rd>   Is the register whose least significant halfword is stored to memory.

<Rn>   Is the register containing the first value used in forming the memory address.

<Rm>   Is the register whose value is added to <Rn> to form the memory address.

Architecture version

All T variants.

Exceptions

Data Abort.

Operation

```
address = Rn + Rm
if address[1:0] == 0
    Memory[address,2] = Rd[15:0]
else
    Memory[address,2] = UNPREDICTABLE
```

Notes

Data Abort   For details of the effects of the instruction if a Data Abort occurs, see *Effects of data-aborted instructions* on page A2-21.

Alignment   If the memory address is not halfword-aligned and no Data Abort occurs, the value written to memory is UNPREDICTABLE

If an implementation includes a System Control coprocessor (see Chapter B3 *The System Control Coprocessor*) and alignment checking is enabled, an address with bit[0] != 0 causes an alignment exception (a type of Data Abort).

Equivalent ARM syntax and encoding

```
STRH   <Rd>, [<Rn>, <Rm>]
```

31 30 29 28	27 26 25 24 23 22 21 20	19 16	15 12	11 8	7 6 5 4	3 2 1 0
1 1 1 0	0 0 0 1 1 0 0 0	Rn	Rd	SBZ	1 0 1 1	Rm

A7.1.67 SUB (1)

15	14	13	12	11	10	9	8		6	5		3	2		0
0	0	0	1	1	1	1	immed_3			Rn			Rd		

SUB (1) (Subtract) subtracts a small constant value from the value of a register and stores the result in a second register.

It updates the condition code flags, based on the result.

Syntax

```
SUB   <Rd>, <Rn>, #<immed_3>
```

where:

<Rd>	Is the destination register for the operation.
<Rn>	Is the register containing the first operand for the subtraction.
<immed_3>	Is a 3-bit immediate value (values 0 to 7) that is subtracted from <Rn>.

Architecture version

All T variants.

Exceptions

None.

Operation

```
Rd = Rn - immed_3
N Flag = Rd[31]
Z Flag = if Rd == 0 then 1 else 0
C Flag = NOT BorrowFrom(Rn - immed_3)
V Flag = OverflowFrom(Rn - immed_3)
```

Equivalent ARM syntax and encoding

```
SUBS  <Rd>, <Rn>, #<immed_3>
```

31 30 29 28	27 26 25 24 23 22 21 20	19 16	15 12	11 10 9 8 7 6 5 4 3	2 0
1 1 1 0	0 0 1 0 0 1 0 1	Rn	Rd	0 0 0 0 0 0 0 0 0	immed_3

A7.1.68 SUB (2)

15	14	13	12	11	10		8	7		0
0	0	1	1	1		Rd			immed_8	
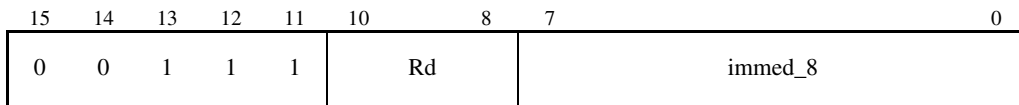
SUB (2) subtracts a large immediate value from the value of a register and stores the result back in the same register.

It updates the condition code flags, based on the result.

Syntax

```
SUB  <Rd>, #<immed_8>
```

where:

<Rd>   Is the register containing the first operand for the subtraction, and is also the destination register for the operation.

<immed_8>   Is an 8-bit immediate value (values 0 to 255) that is subtracted from <Rd>.

Architecture version

All T variants.

Exceptions

None.

Operation

```
Rd = Rd - immed_8
N Flag = Rd[31]
Z Flag = if Rd == 0 then 1 else 0
C Flag = NOT BorrowFrom(Rd - immed_8)
V Flag = OverflowFrom(Rd - immed_8)
```

Equivalent ARM syntax and encoding

```
SUBS  <Rd, <Rd>, #<immed_8>
```

31 30 29 28	27 26 25 24 23 22 21 20	19 16	15 12	11 10 9 8	7 0
1 1 1 0	0 0 1 0 0 1 0 1	Rd	Rd	0 0 0 0	immed_8
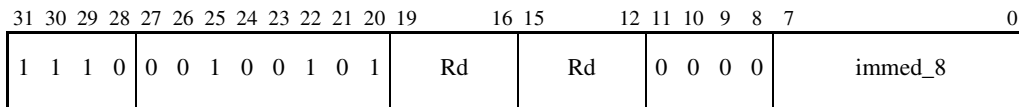
A7.1.69 SUB (3)

15	14	13	12	11	10	9	8		6	5		3	2		0
0	0	0	1	1	0	1		Rm			Rn			Rd	

SUB (3) subtracts the value of one register from the value of a second register and stores the result in a third register.

It updates the condition code flags, based on the result.

Syntax

```
SUB  <Rd>, <Rn>, <Rm>
```

where:

<Rd>	Is the destination register for the operation.
<Rn>	Is the register containing the first operand for the subtraction.
<Rm>	Is the register whose value is subtracted from <Rn>.

Architecture version

All T variants.

Exceptions

None.

Operation

```
Rd = Rn - Rm
N Flag = Rd[31]
Z Flag = if Rd == 0 then 1 else 0
C Flag = NOT BorrowFrom(Rn - Rm)
V Flag = OverflowFrom(Rn - Rm)
```

Equivalent ARM syntax and encoding

```
SUBS  <Rd>, <Rn>, <Rm>
```

31 30 29 28	27 26 25 24 23 22 21 20	19 16	15 12	11 10 9 8 7 6 5 4	3 0
1 1 1 0	0 0 0 0 0 1 0 1	Rn	Rd	0 0 0 0 0 0 0 0	Rm

A7.1.70 SUB (4)

15	14	13	12	11	10	9	8	7	6		0
1	0	1	1	0	0	0	0	1		immed_7	
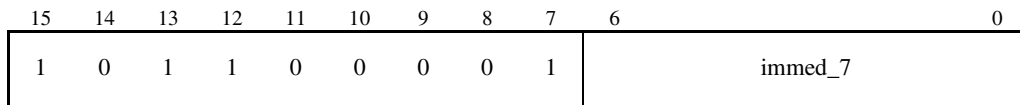
SUB (4) decrements the SP by four times a 7-bit immediate (that is, by a multiple of 4 in the range 0 to 508).

The condition codes are not affected.

Syntax

```
SUB  SP, #<immed_7> * 4
```

where:

SP   Indicates the stack pointer. The result of the operation is also stored in the SP.

<immed_7>   Is a 7-bit immediate value that is multiplied by 4 and then subtracted from the value
                    of the stack pointer.

Architecture version

All T variants.

Exceptions

None.

Operation

```
SP = SP - (immed_7 << 2)
```

Usage

For the Full Descending stack which the Thumb instruction set is designed to use, decrementing the SP is used to allocate extra memory variables on the top of the stack.

Notes

Alternative syntax   This instruction can also be written as SUB SP, SP, #<immed_7> * 4.

Equivalent ARM syntax and encoding

```
SUB  SP, SP, #<immed_7> * 4
```

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 0
1 1 1 0	0 0 1 0 0 1 0 0	1 1 0 1	1 1 0 1	1 1 1 1 0	immed_7

A7.1.71 SWI

15	14	13	12	11	10	9	8	7		0
1	1	0	1	1	1	1	1		immed_8	
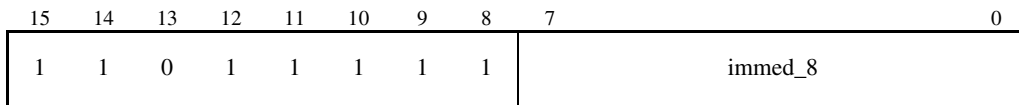
SWI (Software Interrupt) generates a software interrupt or SWI, which is handled by an operating system. See *Exceptions* on page A2-15.

Use it as a call to an operating system service to provide a service.

Syntax

```
SWI  <immed_8>
```

where:

<immed_8>   Is an 8-bit immediate value that is put into bits[7:0] of the instruction. This value is
                   ignored by the processor, but can be used by an operating system's SWI exception
                   handler to determine which operating system service is being requested.

Architecture version

All T variants.

Exceptions

Software Interrupt.

Operation

```
R14_svc  = address of next instruction after the SWI instruction
SPSR_svc = CPSR
CPSR[4:0] = 0b10011              /* Enter Supervisor mode */
CPSR[5]  = 0                     /* Execute in ARM state */
/* CPSR[6] is unchanged */
CPSR[7]  = 1                     /* Disable normal interrupts */
if high vectors configured then
    PC   = 0xFFFF0008
else
    PC   = 0x00000008
```

Equivalent ARM syntax and encoding

```
SWI   <immed_8>
```

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 0
1 1 1 0	1 1 1 1	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	immed_8

A7.1.72 TST

15	14	13	12	11	10	9	8	7	6	5		3	2		0
0	1	0	0	0	0	1	0	0	0		Rm			Rn	

TST (Test) determines whether a particular subset of bits in a register includes at least one set bit. A very common use for TST is to test whether a single bit is set or clear.

It updates the condition code flags, based on the result.

Syntax

```
TST  <Rn>, <Rm>
```

where:

<Rn>	Is the register containing the first operand for the instruction.
<Rm>	Is the register whose value is logically ANDed with the value of <Rn>.

Architecture version

All T variants.

Exceptions

None.

Operation

```
alu_out = Rn AND Rm
N Flag = alu_out[31]
Z Flag = if alu_out == 0 then 1 else 0
C Flag = unaffected
V Flag = unaffected
```

Equivalent ARM syntax and encoding

```
TST  <Rn>, <Rm>
```

31	30	29	28	27	26	25	24	23	22	21	20	19			16	15			12	11	10	9	8	7	6	5	4	3			0
1	1	1	0	0	0	0	1	0	0	0	1		Rn				SBZ			0	0	0	0	0	0	0	0		Rm		

A7.1.73 **UEXT16**

15	14	13	12	11	10	9	8	7	6	5		3	2		0
1	0	1	1	0	0	1	0	1	0	Rm			Rd		

UEXT16 (Unsigned Extend 16) extracts the least significant 16 bits of the operand, and zero extends the value to 32 bits.

Syntax

```
UEXT16 <Rd>, <Rm>
```

where:

<Rd>   Specifies the destination register.

<Rm>   Specifies the operand register.

Architecture version

Version 6 and above.

Exceptions

None.

Operation

```
Rd = Rm AND 0x0000ffff
```

Usage

Use UEXT16 to zero extend a halfword to a word, for example in instruction sequences acting on **unsigned short** values in C/C++.

Equivalent ARM syntax and encoding

```
UUNPK16TO32   <Rd>, <Rm>
```

31	28	27	26	25	24	23	22	21	20	19	18	17	16	15	12	11	10	9	8	7	6	5	4	3	0
Cond		0	1	1	0	1	1	1	1	1	1	1	1	Rd		0	0	0	0	0	1	1	1	Rm	

A7.1.74 UEXT8

15	14	13	12	11	10	9	8	7	6	5		3	2		0
1	0	1	1	0	0	1	0	1	1		Rm			Rd	

UEXT8 (Unsigned Extend 8) extracts the least significant 8 bits of the operand, and zero extends the value to 32 bits.

Syntax

```
UEXT8 <Rd>, <Rm>
```

where:

<Rd>   Specifies the destination register.

<Rm>   Specifies the operand register.

Architecture version

Version 6 and above.

Exceptions

None.

Operation

```
Rd = Rm AND 0x000000ff
```

Usage

Use UEXT16 to zero extend a halfword to a word, for example in instruction sequences acting on **unsigned short** values in C/C++.

Equivalent ARM syntax and encoding

```
UUNPK8TO32   <Rd>, <Rm>
```

31		28	27	26	25	24	23	22	21	20	19	18	17	16	15		12	11	10	9	8	7	6	5	4	3		0
Cond			0	1	1	0	1	1	1	0	1	1	1	1		Rd		0	0	0	0	0	1	1	1		Rm	

A7.2   Thumb instructions and architecture versions

Table A7-1 shows which Thumb instructions are present in each current ARM architecture version that supports Thumb.

Table A7-1 Thumb instructions by architecture

Instruction	v4T, v4TxM	v5T, v5TxM	v6
ADC	Yes	Yes	Yes
ADD (all forms)	Yes	Yes	Yes
AND	Yes	Yes	Yes
ASR (both forms)	Yes	Yes	Yes
B (both forms)	Yes	Yes	Yes
BIC	Yes	Yes	Yes
BKPT	No	Yes	Yes
BL	Yes	Yes	Yes
BLX (both forms)	No	Yes	Yes
BX	Yes	Yes	Yes
CMN	Yes	Yes	Yes
CMP (all forms)	Yes	Yes	Yes
CPS	No	No	Yes
CPY	No	No	Yes
EOR	Yes	Yes	Yes
LDMIA	Yes	Yes	Yes
LDR (all forms)	Yes	Yes	Yes
LDRB (both forms)	Yes	Yes	Yes
LDRH (both forms)	Yes	Yes	Yes
LDRSB	Yes	Yes	Yes
LDRSH	Yes	Yes	Yes
LSL (both forms)	Yes	Yes	Yes

Table A7-1 Thumb instructions by architecture (continued)

Instruction	v4T, v4TxM	v5T, v5TxM	v6
LSR (both forms)	Yes	Yes	Yes
MOV (all forms)	Yes	Yes	Yes
MUL	Yes	Yes	Yes
MVN	Yes	Yes	Yes
NEG	Yes	Yes	Yes
ORR	Yes	Yes	Yes
POP	Yes	Yes	Yes
PUSH	Yes	Yes	Yes
REV (all forms)	No	No	Yes
ROR	Yes	Yes	Yes
SBC	Yes	Yes	Yes
SEXT (both forms)	No	No	Yes
SETEND	No	No	Yes
STMIA	Yes	Yes	Yes
STR (all forms)	Yes	Yes	Yes
STRB (both forms)	Yes	Yes	Yes
STRH (both forms)	Yes	Yes	Yes
SUB (all forms)	Yes	Yes	Yes
SWI	Yes	Yes	Yes
TST	Yes	Yes	Yes
UEXT (both forms)	No	No	Yes

Part B
Memory and System Architectures

Chapter B1
Introduction to Memory and System Architectures

This chapter provides a high-level overview of memory and system architectures. It contains the following sections:

- *About the memory system* on page B1-2
- *Memory hierarchy* on page B1-4
- *L1 cache* on page B1-7
- *L2 cache* on page B1-8
- *Write buffers* on page B1-9
- *Tightly Coupled Memory* on page B1-10
- *Asynchronous exceptions* on page B1-11
- *Semaphores* on page B1-13.

B1.1 About the memory system

The ARM architecture has evolved over the last decade. Over a billion ARM processors have shipped in this period, the vast majority of these ARMv4 or ARMv5 compliant. The memory system requirements of these applications vary considerably, from simple memory blocks with a flat address map, to systems using any or all of the following to optimize their use of memory resources:

- multiple types of memory
- caches
- write buffers
- virtual memory and other memory remapping techniques.

Memory system control has primarily been through a simple cacheable [C] and bufferable [B] control model with additional mechanisms for access privileges. There has been an underlying assumption about memory ordering. In particular, it was assumed that the completion of IO accesses (non-cacheable, non-bufferable - NCNB) occurred in instruction order, with completion signaled by the acknowledge phase of the load or store access.

To meet the demands of higher performance systems and their associated implementations, ARMv6 introduces new disciplines for virtual memory systems and a weakly-ordered memory model including an additional memory barrier command.

Memory behavior is now classified by type:

- strongly ordered
- device
- normal.

These basic types can be further qualified by cacheable, bufferable and shared attributes as well as access mechanisms.

As in the second edition of the ARM ARM, general requirements are described in keeping with the diversity of needs, however, emphasis is given to the ARMv6 virtual memory model and its absolute requirements. The virtual memory support mechanisms associated with earlier variants are described in the backwards compatibility model. Some earlier features are deprecated, and therefore not recommended for use in new designs.

Coprocessor 15 (CP15) remains the primary control mechanism for virtual memory systems, as well as identification, configuration and control of other memory configurations and system features. CP15 provision is a requirement of ARMv6.

The Memory System and Memory Order Model is described in Part B as a series of chapters as follows:

Introduction

This chapter.

Memory Hierarchy

An overview including basic cache theory and the concept of tightly coupled memory.

Memory Order Model

> Memory attributes and order rules introduced with ARMv6.

The System Control Coprocessor

> An overview of the features and support provided.

Virtual Memory System Architecture (VMSA)

> A sophisticated system to control virtual-to-physical address mapping, access permissions to memory, and other memory attributes, based on the use of a *Memory Management Unit* (MMU). The revised ARMv6 model, and the model used by earlier architecture variants, are described.

Protected Memory System Architecture (PMSA)

> An alternative, simpler protection mechanism suitable for many applications that do not require the full facilities provided by the MMU memory system. The revised ARMv6 and earlier architecture variant models are described.

Caches and Write Buffers

> mechanisms provided to control cache and write buffer functionality in a memory hierarchy.

L1 Tightly Coupled Memory Support

> ARMv6 provision including the associated DMA and Smartcache models.

Fast Context Switch Extension

> Describes the Fast Context Switch Extension. This facilitated fast switching between up to 128 processes executing in separate process blocks, each of size up to 32 MB. This is supported in ARMv6 only for backwards compatibility, and its use is deprecated.

——— **Note** ———

Part B describes a wide variety of functionality. ARMv6 is the first architecture variant to standardize the memory model and many system level features. It is the first architecture variant to mandate provision of the System Control Coprocessor, and a level of consistency at the system level for hardware and software design. Because of this, ARMv6 is considered a watershed in terms of how material is presented in Part B. Absolute requirements are provided for ARMv6 compliant implementations, whereas information can only be considered as system guidelines for earlier architecture variants.

It is assumed that all versions of the architecture prior to version 4 are now OBSOLETE. For example, all references to 26-bit mode have been removed.

Some ARM processors prior to ARMv6 have implemented functions in a different manner from those described here. Because of this, the datasheet or Technical Reference Manual for a particular ARM processor is the definitive source of information for memory and system control facilities. Processors which have followed the guidelines are more likely to be compatible with existing and future ARM software. ARMv6 establishes a baseline for system design, but there will always be additional functionality and areas of implementation dependent options. The system designer is strongly encouraged to read the architecture in conjunction with vendor datasheets for optimal system design and performance.

B1.2 Memory hierarchy

Good system design is a balance of many trade-offs to achieve the overall system performance and cost goals. An important part of this decision process is the memory provision:

- types of memory, for example ROM, Flash, DRAM, SRAM, disk based storage
- size - capacity and silicon area
- access speed - core clock cycles required to read or write a location
- architecture - Harvard (separate instruction and data memories) or Von Neumann (unified memory).

As a general rule, the faster the memory access time, the more constrained the amount of resource available, because it needs to be closely coupled to the processor core, that is, on the same die. Even on-chip memory may have different timing requirements because of its type or size, power constraints, and the associated critical path lengths to access it in the physical layout. Caches provide a means to share the fastest, most expensive system memory resources between the currently active process threads in an application.

Where a system is designed with different types of memory in a layered model, this is referred to as a *memory hierarchy*. Systems can employ caches at multiple levels. The outer layers trade increased latency for increasing size. All the caches in the system must adhere to a memory coherency policy, which is part of the system architecture. Such layered systems usually number the layers - level 1, level 2 ... level *n*- with the increasing numbers representing increased access times for layers further from the core.

IO can also be provided at the different layers, that is, some no-wait-state register-based peripherals at level 1, out to memory mapped peripherals on remote system buses.

Figure B1-1 shows an example memory hierarchy.
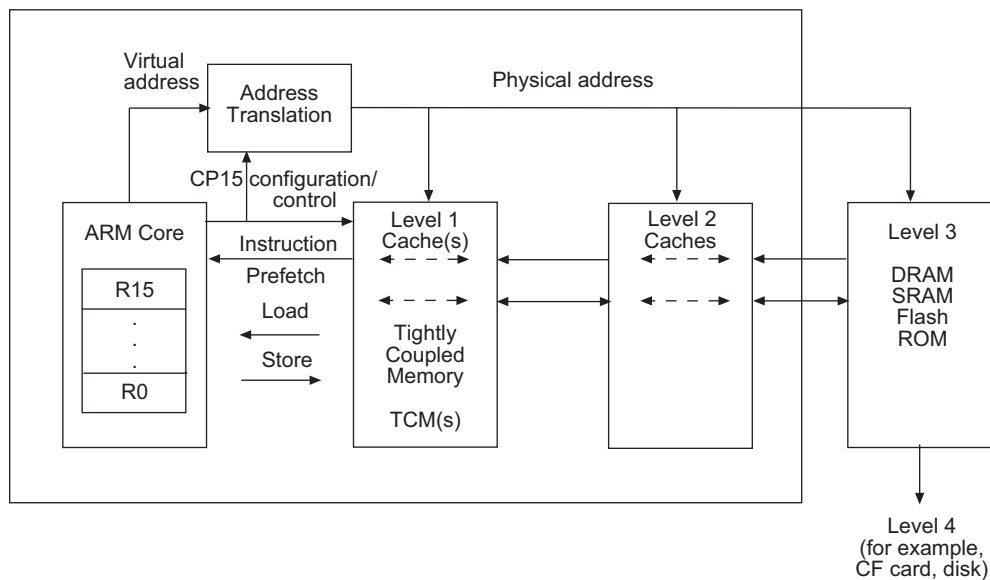
Figure B1-1 Memory hierarchy example

The ARMv6 specifies the *Level 1* (L1) subsystem, providing cache, *Tightly-Coupled Memory* (TCM), and an associated TCM-L1 DMA system. The architecture permits a range of implementations, with software visible configuration registers to allow identification of the resources that exist. Options are provided to support the L1 subsystem with a Memory Management Unit (VMSAv6) or a simpler Memory Protection Unit (PMSAv6).

Some provision is also made for multiprocessor implementations and Level 2 (L2) caches. However, these are not fully specified in this document. To ensure future compatibility, it is recommended that Implementors of L2 caches and closely-coupled multiprocessing systems work closely with ARM.

VMSAv6 describes Inner and Outer attributes which are defined for each page-by-page. These attributes are used to control the caching policy at different cache levels for different regions of memory. Implementations can use the Inner and Outer attributes to describe caching policy at other levels in an IMPLEMENTATION DEFINED manner. See sections *Memory region attributes* on page B4-11 for the architecture details. All levels of cache need appropriate cache management and must support:

*   cache cleaning (write-back caches only)
*   cache invalidation (all caches).

ARM processors and software are designed to be connected to a byte-addressed memory. Prior to ARMv6, addressing was defined as word invariant. Word and halfword accesses to the memory ignored the byte alignment of the address, and accessed the naturally-aligned value that was addressed, that is, a memory access ignored address bits 0 and 1 for word access, and ignored bit 0 for halfword accesses. The endianness of the ARM processor normally matched that of the memory system, or was configured to match it before any non-word accesses occurred.

ARMv6 introduces:

*   a byte-invariant address model
*   support of unaligned word and halfword accesses
*   additional control features for loading and storing data in a little or big endian manner.

See section *Endian Support* on page A2-29 for details.

B1.2.1 Managing the memory hierarchy

Attributes must be set to mark memory that is used to hold programs and data as follows:

Main (RAM) memory

   normally set as cacheable and bufferable

ROM memory   normally set as cacheable, and marked as read only. The bufferable attribute is not normally used and is set to 1.

Frame buffers can be cacheable, but frame buffers on write-back cache implementations must be copied back to memory after the frame buffer has been updated. Frame buffers can be bufferable, but the write buffer must be written back to memory after the frame buffer has been updated.

ARM processors do not normally support cache coherence between the ARM and other system bus masters. Bus snooping is currently not supported in the architecture. If memory data is to be shared between multiple bus masters without taking special software measures to ensure coherency, the data must be mapped as:

• uncacheable to ensure that all reads access main memory

• unbufferable to ensure that all writes access main memory.

Alternatively, using software, you can manage the coherence of data buffers for multi-master access by:

• cleaning data from write-back caches and write buffers to memory when the processor has written to the data buffer and before the other bus master reads the buffer

• flushing the relevant data from caches when the buffer is being read after the other bus master has written the buffer.

Uncached, unbuffered semaphores can be used to maintain synchronization between multiple bus masters (see *Semaphores* on page B1-13 for more information).

For implementations with write-back caches, all dirty cache data must be written back before any alterations are made to the MMU page tables. Similarly any page table updates which are cacheable must be written back to memory before the revised entries are used. The former is to ensure valid address translation for all clean operations, and the latter is to ensure that page table lookups return the correct data; page table lookups may, but do not have to, monitor the data caches.

Harvard architecture implementations - supporting separate instruction and data caches - may not maintain coherence between the data and instruction caches. If the instruction stream is written, the instruction cache and data cache must be made coherent. This can be achieved by the following:

• cleaning the data cache, that is, storing dirty data to memory

• draining the write buffer, that is, completing all buffered writes

• flushing the instruction cache.

Instruction and data memory incoherence occurs after a program has been loaded (and therefore treated as data) and is about to be executed. It also occurs if self-modifying code is used or generated.

B1.3 L1 cache

Before ARMv6, ARM caches were normally implemented as virtual caches, with virtual indexing and virtual address tags. With this model, physical pages were only mapped into a single virtual page, otherwise the result was UNPREDICTABLE. These implementations did not provide coherence between multiple virtual copies of a single physical page.

ARMv6 specifies a cache architecture where the expected behavior is that normally associated with physically tagged caches. The ARMv6 L1 cache architecture is designed to reduce the requirement for cache clean and/or invalidation on a context switch. Flexibility on the size, associativity or organization of the caches within this subsystem is provided in the Coprocessor System Control Register (CP15). The cache organization may be a Harvard architecture with separate instruction and data caches, or a von Neumann architecture with a single, unified cache.

In a Harvard architecture, an implementation does not need to include hardware support for coherency between the Instruction and Data caches. Where such support would be required, for example, in the case of self-modifying code, the software must make use of the cache cleaning instructions to avoid such problems.

An ARMv6 L1 cache must appear to software to behave as follows:

• the entries in the cache do not need to be cleaned and/or invalidated by software for different virtual to physical mappings

• aliases to the same physical address may exist in memory regions that are described in the page tables as being cacheable, subject to the restrictions for 4KB small pages outlined in section TBD.

Caches can be implemented with virtual or physical addressing (including indexing) provided these behavior requirements are met. ARMv6 L1 cache management uses virtual addresses, which is consistent with earlier architecture guidelines and implementations.

For architecture details on the L1 cache see Chapter B6 *Caches and Write Buffers*.

B1.4   L2 cache

L1 caches are always tightly coupled to the core, but L2 caches can be either:

*   tightly coupled to the core
*   implemented as memory mapped peripherals on the system bus.

A recommended minimum set of L2 cache commands is defined for configuration and control. Closely-coupled L2 caches must be managed through the System Control Coprocessor. It is IMPLEMENTATION DEFINED whether they use virtual or physical addresses for control functions. Memory mapped L2 caches must use physical address based control.

Further levels of cache are possible, but their control is not mandated within ARMv6 except that they must comply with:

*   the inner and outer attribute model described in *Memory region attributes* on page B4-11.

*   coherency needs associated with managing multi-level caches through the System Control Coprocessor interface, see *Considerations for additional levels of cache* on page B6-11.

For architecture details on the L2 cache see section *L2 cache*.

B1.5   Write buffers

Write buffers are highly desirable for performance reasons. They allow a core to retire an instruction in a write buffer, rather than having to wait until a potentially slow memory access completes. This assumes that it is acceptable to delay the memory mapped write in this manner.

Some ARM implementations incorporate a merging write buffer that coalesces multiple writes to the same cache line into a single write to main memory. This may cause multiple writes to one location to be merged, writes to be re-ordered to memory, or a combination of both. Therefore, I/O locations must not normally be marked as bufferable, to ensure all writes are issued to the I/O device in the correct order.

For writes to bufferable areas of memory, precise aborts can only be signalled to the processor as a result of conditions that are detectable at the time the data is placed in the write buffer. Conditions that can only be detected when the data is later written to main memory, such as an ECC error from main memory, must be handled by other methods, by raising an interrupt or an imprecise abort.

B1.6   Tightly Coupled Memory

The Tightly-Coupled Memory (TCM) is an area of memory that can be implemented alongside the L1 cache, as part of the level 1 memory subsystem. The TCM is physically addressed, with each bank occupying a unique part of the physical memory map. See section B1.TBD_smartcache for an optional, smartcache, ARMv6 usage model. In keeping with the L1 cache, the TCM may be structured as a Harvard architecture with separate instruction and data TCM, or as a Von Neumann architecture with a unified TCM.

The TCM is designed to provide low latency memory that can be used by the processor without the unpredictability that is a feature of Caches. Such memory can be used to hold critical routines, such as interrupt handling routines or real-time tasks, where the indeterminacy of a cache would be highly undesirable. Other example uses are:

- scratchpad data
- data types whose locality properties are not well suited to caching
- critical data structures such as Interrupt stacks.

For architectural details on Tightly Coupled Memory, see Chapter B_TBD_TCM.

B1.6.1   Tightly Coupled Memory versus cache memory

The TCM is designed to be used as part of the physical memory map of the system, and is not expected to be backed by a level of external memory with the same physical addresses. For this reason, the TCM behaves differently from the caches for regions of memory which are marked as being Write-Through cacheable. In such regions, no external writes occur in the event of a write to memory locations contained in the TCM.

It is an architectural requirement that memory locations are contained either in the TCM or the cache, not in both. In particular, no coherency mechanisms are supported between the TCM and the cache. This means that it is important when allocating the base address of the TCM to ensure that the TCM address range does not overlap with any valid cache entries.

B1.6.2   DMA support for Tightly Coupled Memory

ARMv6 includes a DMA model with register support for its configuration. This is the only mechanism other than the associated processor core that can read and write the TCM. Up to two DMA channels are provided for. This allows chained operations, see section B_TBD_L1_DMA for architectural details.

——— **Note** ———

The TCM DMA mechanism and *smartcache* functionality described in *SmartCache Behavior* on page B7-5 are mutually exclusive.

B1.7   Asynchronous exceptions

Many exceptions are synchronous events related to instruction execution in the core, however the following exceptions cause asynchronous events to occur:

- *Reset*
- *Interrupts*
- *Imprecise aborts* on page B1-12.

B1.7.1   Reset

This is the only non-maskable event in the ARM architecture. See section A_TBD_program_model_reset for more information.

B1.7.2   Interrupts

ARM processors implement fast and normal levels of interrupt. Both interrupts are signalled externally, and many implementations synchronize interrupts before an exception is raised.

***Fast interrupt request* (FIQ)**

Disables subsequent normal and fast interrupts by setting the I and F bits in the CPSR.

***Normal interrupt request* (IRQ)**

Disables subsequent normal interrupts by setting the I bit in the CPSR.

Vectored Interrupts

(IMPLEMENTATION DEFINED whether IRQ or FIQ or some of each.)

Some implementations incorporate a mechanism controlled by the System Control Coprocessor to return interrupt vectors directly to the core. The mechanism typically applies to the IRQ mode, but can also apply to FIQ mode. The exact behavior is IMPLEMENTATION DEFINED.

For more information on interrupts, see sections A_TBD_prog_model_Interrupts. (&_TBD_Veinterrupts?)

Cancelling interrupts

It is the responsibility of software (the interrupt handler) to ensure that the cause of an interrupt is cancelled (no longer signalled to the processor) before interrupts are re-enabled (by clearing the I or F bit, or both, in the CPSR). Interrupts can be cancelled with any instruction that might make an external data bus access, that is:

- any load
- any store
- a swap
- any coprocessor instruction.

Cancelling an interrupt via an instruction fetch is UNPREDICTABLE.

Cancelling an interrupt with a load multiple that restores the CPSR and re-enables interrupts is UNPREDICTABLE.

Devices that do not instantaneously cancel an interrupt (that is, they do not cancel the interrupt before letting the access complete) must be probed by software to ensure that interrupts have been cancelled before interrupts are re-enabled. This allows a device connected to a remote I/O bus to operate correctly. The polled model is the architected model for ARMv6 systems. See section B_TBD_mem_order_model for more information.

B1.7.3 Imprecise aborts

ARMv6 has introduced the concept of imprecise aborts. These aborts can occur after the instruction which caused the abort has been retired. Therefore an imprecise abort is fatal, at least to the process that caused it, or requires external resources to record address, data and control information for a software recovery. These aborts are masked on entry to most exception vectors, and can be masked by privileged software using the CPSR_A bit. See section A_TBD_exceptions for more information.

B1.8   Semaphores

The Swap and Swap Byte instructions have predictable behavior when used in either of two ways:

- Systems with multiple bus masters that use the Swap instructions to implement semaphores to control interaction between different bus masters.

    In this case, the semaphores must be placed in an uncached and unbufferable region of memory. The Swap instruction then causes a locked read-write bus transaction.

    This type of semaphore can be externally aborted.

- Systems with multiple threads running on a uniprocessor that use the Swap instructions to implement semaphores to control interaction of the threads.

    In this case, the semaphores can be placed in a cached and bufferable region of memory, and a locked read-write bus transaction might or might not occur. The Swap and Swap Byte instructions are likely to have better performance on such a system than they do on a system with multiple bus masters (as described above).

    This type of semaphore has UNPREDICTABLE behavior if it is externally aborted.

From ARMv6, load and store exclusive instructions (LDREX and STREX) are the preferred method of implementing semaphores for system performance reasons. The new mechanism is referred to as synchronizing primitives, and requires data monitor logic within the memory system that monitors access to the requested location from all sources in the shared memory model case. The instructions provide a degree of decoupling between the load and store elements, with the store only being successful if no other resource has written to the location since its associated load. See section *Synchronization primitives* on page A2-48 for more details.

Semaphores using the SWAP/SWAPB mechanism placed in uncacheable/bufferable memory regions have UNPREDICTABLE results. Similarly, Semaphores placed in cacheable/unbufferable memory regions have UNPREDICTABLE results, unless using the synchronous primitives mechanism, and all resources access the semaphores through the same shared cache resource.

Introduction to Memory and System Architectures

Chapter B2
Memory Order Model

This chapter provides a high-level overview of the memory order model. It contains the following sections:

- *About the memory order model* on page B2-2
- *Memory classes* on page B2-3
- *Memory attributes and the memory model* on page B2-9
- *Ordering requirements for memory accesses* on page B2-16.

B2.1 About the memory order model

The architecture prior to version 6 did not attempt to define the acceptable memory ordering of explicit memory transactions, describing the regions of memory according to the hardware approaches that had previously been used to implement such memory systems. Thus regions of memory had been termed as being one of *Write-Through Cacheable*, *Write-Back Cacheable*, *Non-Cacheable Bufferable* or *Non-Cacheable, Non-Bufferable*. These terms are based on the previous hardware implementations of cores and the exact properties of the memory transactions could not be rigorously inferred from the memory names. Implementations have chosen to interpret these names in different ways, leading to potentially incompatible uses.

In a similar manner, the order in which memory accesses could be presented to memory was not defined, and in particular there was no definition of what order could be relied upon by an observer of the memory transactions generated by a processor. As implementations and systems become more complicated, these undefined areas of the architecture move from being simply based on a standard default to having the potential of presenting significant incompatibilities between different implementations; at processor core and system level.

Version 6 introduces a set of memory types with described memory access properties, which have been made to fit in a largely backwards compatible manner to the defacto meanings of the original memory regions. A potential incompatibility has been introduced with the need for a software polling policy when it is necessary for the program to be aware that memory accesses to I/O space have completed, and all side-effects are visible across the whole system. This reflects the increasing difficulty of ensuring linkage between the completion of memory accesses and the execution of instructions within a complex high-performance system.

A memory attribute to indicate whether a region of memory is shared between multiple processors (and therefore requires an appearance of cache transparency in an ordering model) is also introduced. Implementations remain free to chose the mechanisms to implement this functionality.

B2.2 Memory classes

All memory accesses can be considered as conforming to two classes of access:

*   *Memory-like memory*
*   *I/O-like memory*

B2.2.1 Memory-like memory

Memory-like memory is idempotent, and exhibits the following properties:

*   Write transactions can be repeated with no side effects.

*   Repeated read transactions return the last value written to the resource being read.

*   Transactions can be restarted if interrupted.

*   Multibyte accesses need not be atomic, and can be restarted or replayed. Unaligned accesses are supported.

*   Unaligned accesses are supported.

*   Transactions can be merged prior to accessing the target memory system.

*   Read transactions can prefetch additional memory locations with no side effects.

B2.2.2 I/O-like memory

I/O-like memory is typically used to map system peripherals. The access might be to a volatile location, or might cause side effects into other memory or peripherals. Examples of I/O accesses are:

*   FIFOs where consecutive accesses add (write) or remove (read) queued values

*   interrupt controller registers where an access might be used as an interrupt acknowledge changing the state of the controller itself

*   Memory controller configuration registers that are used to set up the timing (and correctness) of areas of normal memory.

To ensure system correctness, I/O-like memory accesses are more restrictive than those to normal memory:

*   accesses (reads and writes) may have side affects
*   transactions must not be repeated, for example, on return from an exception
*   transaction size and order must be maintained.

B2.2.3   Memory access restrictions

The following restrictions apply to memory accesses:

* For any access X, the bytes accessed by X must either all be in memory-like memory or all in I/O-like memory. Otherwise, the behavior of the access is UNPREDICTABLE. That is. unaligned accesses that span a boundary between I/O-like and memory-like memory are UNPREDICTABLE.

* For any two memory accesses X and Y such that X and Y are generated by the same instruction, X and Y must either both be memory-like or both be I/O-like. Otherwise, the results are UNPREDICTABLE. That is, an LDC, LDM, LDRD, STC, STM, or STRD that spans a boundary between I/O-like and memory-like is UNPREDICTABLE.

* Instructions that generate unaligned memory accesses to I/O-like memory are UNPREDICTABLE.

* For instructions that generate accesses to I/O-like memory, processor core implementations do not change the sequence of accesses specified by the instruction's pseudo-code. This includes not changing how many accesses there are, nor their time order, nor the data sizes and other properties of each individual access. Furthermore, processor core implementations expect any attached memory system to be able to identify accesses as memory-like or I/O-like and to obey similar restrictions with regard to the number, time order, data sizes and other properties of the accesses.

* Exceptions to this rule are:

    — An implementation of a processor core can break this rule, provided that the information it does supply to the memory system enables the original number, time order, and other details of the accesses to be reconstructed. In addition the implementation must place a requirement on attached memory systems to do this reconstruction when the accesses are to I/O-like memory.

        For example, the word loads generated by an LDM might be paired into 64-bit accesses by an implementation with a 64-bit bus. This is because the instruction semantics ensure that the 64-bit access is always a word load from the lower address followed by a word load from the higher address, provided a requirement is placed on memory systems to unpack the two word loads if the memory is I/O-like.

    — Any implementation technique that produces results that cannot be observed to be different from those described above is legitimate.

Programmers need to rely upon the number, time order, and other details of memory accesses made to I/O-like memory by a multi-access instruction. It is IMPLEMENTATION DEFINED whether multi-access instructions behave correctly in fast interrupt configurations.

── **Note** ──

Prior to ARMv6 it is IMPLEMENTATION DEFINED whether a fast interrupt mode is supported. From ARMv6, Fast Interrupt (low interrupt latency) support is controlled from the System Control coprocessor (FI-bit).

* Multi-access instructions that load or store R15 must only access memory-like memory. If they access I/O-like memory the results are UNPREDICTABLE.

- Instruction fetches must only access memory-like memory. If they access I/O-like memory, the results are UNPREDICTABLE.

B2.3   Read and write definitions

Memory accesses can be either reads or writes.

B2.3.1   Reads

Reads are defined as memory operations that have the semantics of a load.

In the ARM instruction set, these are:
*   `LDM, LDRH, LDRSH, LDRB, LDRSB`
*   `LDM, LDRD, LDRT, LDRBT,`
*   `LDC, RFE, SWP, SWPB, LDREX, STREX.`

In the Thumb instruction set, they are:
*   `LDR, LDRH, LDRSH, LDRB, LDRSB`
*   `LDM, POP.`

Java instructions that are accelerated by hardware can cause a number of reads to occur, according to the state of the Java stack and the implementation of the Java hardware acceleration.

ARM DDI 0100F-02

B2.3.2   Writes

Writes are defined as operations that have the semantics of a store.

In the ARM instruction set, these are:

*   STR, STRH, STRB
*   STM, STRD, STRT, STRBT
*   STC, SRS, SWP, SWPB, STREX

In the Thumb instruction set, they are:

*   STR, STRH, STRB
*   STM, PUSH

Java instructions that are accelerated by hardware can cause a number of writes to occur, according to the state of the Java stack and the implementation of the Java hardware acceleration.

B2.3.3   Write completion

Due to memory hierarchy, the memory system may indicate to the CPU that a write to Normal memory has been "completed" at a point in the transaction where the memory system is unable to guarantee that the effect of the write is visible to all processors in the system. This may be due to the write terminating in a write-back cache, or due to being temporarily held in a write buffer. Explicit software action may be necessary where the write needs to be synchronized with events in a wider context. This may involve cache cleaning operations, forced flushing of a write buffer, or both.

Device or Strongly Ordered memory is by definition non-cacheable, however, Device memory may be buffered (defined as shared), as long as the rules on transaction size and ordering of Device memory accesses are maintained.

Synchronization primitives are a special case, where the write must complete at (or beyond) the point of synchronization. The supporting data monitor may be associated with a cache or memory in the system hierarchy, and must be at a point in the system visible to all sharing agents.

Memory barriers can be used to force a synchronization point between different types of access as explained in sections TBD_B2.8 and TBD_B2.9.

B2.3.4   Memory synchronization primitives

The memory synchronization primitive instructions are defined as those instructions that are used to ensure memory synchronization:

*   LDREX, STREX
*   SWP, SWPB.

Memory Synchronization Primitives exist to ensure synchronization between different processes, that may be running on the same processor, or on different processors. Prior to ARMv6, support consisted of the SWP and SWPB instructions. ARMv6 has introduced new LDREX and STREX (Load and Store Exclusive) instructions.

You can use memory synchronization primitives in regions of memory marked as shared or non-shared memory, see sections B_TBD_shared_norm_mem and B_TBD_nonshared_norm_mem. You can use synchronization primitives with non-shared memory when the processes to be synchronized are running on the same processor, if they use shared areas of memory and the processes to be synchronized are running on different processors.

See *Synchronization primitives* on page A2-48 for the architecture details.

Synchronization primitives are required to ensure correct operation of system semaphores within the memory order model.

ARM DDI 0100F-02

B2.4   Memory attributes and the memory model

Traditionally, all memory has been tagged with a combination of two control bits in ARM's virtual and protected memory management models, VMSA and PMSA respectively. The bits are:

* a bufferable (B) bit (allow write buffering between the core and memory)

* a cacheable (C) bit.

These are traditionally interpreted to define the memory behavior of a given location as shown in Table B2-1.

Table B2-1 Interpretation of cacheable and Bufferable bits

C	B	Write-through cache	Write-back only cache	Write-back/write-through cache
0	0	Uncached/unbuffered	Uncached/unbuffered	Uncached/unbuffered
0	1	Uncached/buffered	Uncached/buffered	Uncached/buffered
1	0	Cached/unbuffered	UNPREDICTABLE	Write-through cached/buffered
1	1	Cached/buffered	Cached/buffered	Write-back cached/buffered

ARMv6 has introduced a more closely defined memory model. Backwards compatibility is described in section PartB_TBD_MemOrderModel_v6_Back_Compatability.

B2.4.1   ARMv6 memory types and associated attributes

ARMv6 provides a set of memory attributes that have characteristics designed to provide for particular devices, including memory devices, that are contained in the memory map. The ordering of accesses for regions of memory is also defined by the memory attributes.

There are three mutually exclusive main memory type attributes to describe the memory regions:
* Strongly Ordered
* Device
* Normal.

If the same memory location is marked as having two different attributes in a Memory Management Unit, for example by the use of synonyms in a virtual to physical address mapping, the behavior is UNPREDICTABLE.

The Shared attribute indicates whether the memory is private to a single processor, or accessible from multiple processors or other bus master resources, for example an intelligent peripheral with DMA capability.

─── **Note** ───

Strongly Ordered and Device types are primarily used with I/O-like (device) memory, but can be used with memory-like memory. The Normal memory attribute is used exclusively with memory-like (normal) memory.

Table B2-2 shows a summary of the memory attributes.

Table B2-2 Memory attribute summary

Memory type attribute	Shared attribute	Other attributes	Description
Strongly Ordered	-		All memory accesses to Strongly Ordered memory occur in program order. Some backwards compatibility constraints exist with ARMv5 instructions which change the CPSR interrupt masks (see section TBD_strongly_ordered). All Strongly Ordered accesses are assumed to be Shared.
Device	Shared		Designed to handle memory mapped peripherals that are shared by several processors.
	Non-Shared		Designed to handle memory mapped peripherals that are used only by a single processor.
Normal	Shared	Non-cacheable/ Write-Through cacheable/ Write-Back cacheable	Designed to handle normal memory which is shared between several processors
	Non-Shared	Non-cacheable/ Write-Through cacheable/ Write-Back cacheable	Designed to handle normal memory which is used only by a single processor

B2.4.2   Normal memory attribute

The Normal memory attribute is defined as being distinct from Device memory attribute as described in *Device memory attribute* on page B2-12, and the Strongly Ordered memory attribute described in *Strongly Ordered memory attribute* on page B2-14. This attribute is defined for each page in an MMU. It is designed to provide memory access orderings that are suitable for *normal* memory. Such memory stores information without side effects. Normal memory may be writeable or read-only.

For writeable *normal* memory unless there is a change to the physical address mapping:

*   A load from a specific location will return the most recently stored data at that location for the same processor.

*   Two loads from a specific location, without a store in between, will return the same data for each load.

For read-only *normal* memory:

*   Two loads from a specific location will return the same data for each load.

This behavior describes most memory used in a system. The term memory-like is used to describe this sort of memory. For the purpose of this section, writeable *normal* memory and read-only *normal* memory are not distinguished.

Regions of memory with the Normal attribute can be further defined as being Shared or Non-Shared, for each page in an MMU. If the same memory location is marked as Shared Normal and Non-Shared Normal in a MMU, for example by the use of synonyms in a virtual to physical address mapping, UNPREDICTABLE behavior results.

All explicit accesses to memory marked as Normal must correspond to the ordering requirements of accesses described in section TBD_Mem_order. Accesses to Normal Memory conform to the Weakly Ordered model of memory ordering. A description of this model can be found in standard texts describing memory ordering issues. A recommended text is chapter 2 of *Memory Consistency Models for Shared Memory-Multiprocessors*, Kourosh Gharachorloo, Stanford University Technical Report CSL-TR-95-685.

Non-shared Normal memory

The Non-Shared Normal memory attribute is designed to describe normal memory that can be accessed only by a single processor.

A region of memory marked as Non-Shared Normal does not have any requirement to make the effect of a cache transparent.

Shared Normal memory

The Shared Normal memory attribute is designed to describe normal memory that can be accessed by multiple processors or other system masters.

A region of memory marked as Shared Normal is one in which the effect of interposing a cache (or caches) on the memory system is entirely transparent. Implementations can use a variety of mechanisms to support this, from very simply not caching accesses in shared regions to more complex hardware schemes for cache coherency for those regions.

In implementations with a TCM, the regions of memory covered by the TCM must not be marked as Shared. Marking an area of memory covered by the TCM as being Shared can result in UNPREDICTABLE behavior.

Writes to Shared Normal Memory may not be atomic, that is, all observers might not see the writes occurring at the same time. To preserve coherence where two writes are made to the same location, it is required that the order of those writes is seen to be the same by all observers. Reads to Shared Normal Memory that are aligned in memory to the size of the access are atomic.

Cacheable write-through, cacheable write-back and non-cacheable memory

In addition to marking a region of normal memory as being Shared or Non-Shared, each page of memory marked in an MMU as Normal can also be marked as being one of:

*   cacheable write-through
*   cacheable write-back
*   non-cacheable.

This marking is independent of the marking of a region of memory as being Shared or Non-Shared. It indicates the required handling of the data region for reasons other than those to handle the requirements of shared data. As a result, it is acceptable for a region of memory that is marked as being cacheable and shared not to be held in the cache in an implementation which handles shared regions as not caching the data.

If the same memory locations are marked as having different Cacheable attributes, for example by the use of synonyms in a virtual to physical address mapping, UNPREDICTABLE behavior results.

B2.4.3 Device memory attribute

The Device memory attribute is defined for memory locations where an access to the location might cause side effects, or where the value returned for a load might vary depending on the number of loads performed. Memory mapped peripherals and I/O locations are typical examples of areas of memory that should be marked as being Device. The Device attribute is defined for each page in an MMU.

Accesses to memory mapped locations that have side effects that apply to memory locations that are normal memory can require Memory Barriers to ensure correct execution. An example where this could be an issue is the programming of the control registers of a memory controller while accesses are being made to the memories controlled by the controller.

Instruction fetches must not be performed to areas of memory containing read sensitive devices, because there is no ordering requirement between instruction fetches and explicit accesses. As a result, instruction fetches from such devices can result in UNPREDICTABLE behavior. Implementations can prefetch up to 64

bytes sequentially ahead of the current instruction being executed, so read sensitive devices must be located in memory to allow for such prefetching. Implementations must ensure that prefetching down non-sequential paths, for example as a result of a branch predictor, cannot cause unwanted accesses to read sensitive devices.

Explicit accesses from the processor to regions of memory marked as Device occur at the size and order defined by the instruction. The number of accesses that occur to such locations is the number that is specified by the program. Implementations must not repeat accesses to such locations when there is only one access in the program, that is, the accesses are not *restartable*. An example where an implementation may wish to repeat an access is before and after an interrupt, in order to allow the interrupt to cause a slow access to be abandoned. Such implementation optimizations must not be performed for regions of memory marked as Device.

If a memory operation that causes multiple transactions, for example an LDM or an unaligned memory access, crosses a 4KB address boundary, it may perform more accesses than are specified by the program regardless of one or both of the areas being marked as Device. For this reason, accesses to volatile memory devices must not be made using single instructions that cross a 4KB address boundary. This restriction is expected to cause restrictions to the placing of such devices in the memory map of a system, rather than to cause a compiler to be aware of the alignment of memory accesses.

In addition, address locations marked as Device are not held in a cache. While writes to device memory may be buffered, writes shall only be merged where the correct number of accesses, order, and their size is maintained.

Shared attribute

Regions of Memory marked as Device are distinguished by the Shared attribute in the MMU. These can be referred to as:

• memory marked as Shared Device

• memory marked as Non-Shared Device.

Explicit accesses to memory with each of the sets of attributes occur in program order relative to other explicit accesses to memory with the same set of attributes.

All explicit accesses to memory marked as Device must correspond to the ordering requirements of accesses described in section *Ordering requirements for memory accesses* on page B2-16.

If the same memory location is marked as being Shared Device and Non-Shared Device in an MMU, for example by the use of synonyms in a virtual to physical address mapping, UNPREDICTABLE behavior results.

An example of an implementation in which the Shared attribute is used to distinguish memory accesses is an implementation that supports a local bus for its private peripherals, whereas system peripherals are situated on the main system bus. Such a system might have more predictable access times for local peripherals such as watchdog timers or interrupt controllers.

For shared device memory, the data of a write is visible to all observers before the end of a DataWriteBarrier memory barrier. For non-shared device memory, the data of a write is visible to the processor before the end of a DataWriteBarrier memory barrier (see *Explicit Memory Barriers* on page B2-32).

B2.4.4   Strongly Ordered memory attribute

The Strongly Ordered memory attribute is defined for each page in the MMU. Accesses to memory marked as Strongly Ordered have a strong memory-ordering model for all explicit memory accesses from that processor. An access to memory marked as Strongly Ordered acts as a memory barrier to all other explicit accesses from that processor, until the point at which the access is complete, that is, it has changed the state of the target location or data has been returned. In addition, an access to memory marked as Strongly Ordered must complete before the end of a Memory Barrier (see *Memory Barriers* on page B2-29).

To maintain backwards compatibility with ARMv5, any ARMv5 instructions that implicitly or explicitly change the interrupt masks in the CSPR and appear in program order after a Strongly Ordered access must wait for the Strongly Ordered memory access to complete. These instructions are MSR, with the control field mask bit set, and the flag-setting variants of arithmetic and logical instructions with R15 as the destination register (these copy the SPSR to CSPR). This requirement exists only for backwards compatibility with previous versions of the ARM architecture, and the behavior is deprecated in ARMv6. Programs must not rely on this behavior, but instead include an explicit Memory Barrier between the memory access and the following instruction, see *Memory Barriers* on page B2-29.

Explicit accesses from the processor to memory marked as Strongly Ordered occur at their program size, and the number of accesses that occur to such locations is the number that are specified by the program. Implementations must not repeat accesses to such locations when there is only one access in the program, that is, the accesses are not *restartable*.

If a memory operation which causes multiple transactions, such as an LDM or an unaligned memory access, crosses a 4KB address boundary, then it may perform more accesses than are specified by the program regardless of one or both of the areas being marked as Strongly Ordered. For this reason, it is important that accesses to volatile memory devices are not made using single instructions that cross a 4KB address boundary.

Address locations marked as Strongly Ordered are not held in a Cache, and are treated as Shared memory locations.

For Strongly Ordered memory, the data and side-effects of a write is visible to all observers before the end of a DataWriteBarrier memory barrier (see *Explicit Memory Barriers* on page B2-32).

B2.4.5 Backwards compatibility

ARMv6 memory attributes are significantly different from those in previous versions of the architecture. Table B2-3 shows the interpretation of the earlier memory types in the light of this definition.

Table B2-3 Backwards compatibility

Previous architectures	ARMv6 attribute
NCNB (Non-cacheable, Non-Bufferable)	Strongly Ordered [a]
NCB (Non-cacheable, Bufferable)	Shared Device [a]
Write-Through cacheable, Bufferable	Non-Shared Normal (Write-Through cacheable)
Write-Back cacheable, Bufferable	Non-Shared Normal (Write-Back cacheable)

a. Memory locations contained within the TCMs are treated as being Non-Cacheable, not Strongly Ordered or Shared Device

B2.5   Ordering requirements for memory accesses

ARMv6 defines access restrictions in the memory ordering allowed, depending on the memory attributes of the accesses involved. Figure B2-1 shows the memory ordering between two explicit accesses A1 and A2, where A1 occurs before A2 in program order.

The symbols used in Table 1 are as follows:

<   Accesses must occur strictly in program order, that is. A1 must occur strictly before A2. It must be impossible to tell otherwise from observation of the read/write values and side effects caused by the memory accesses.

(blank)   Accesses can occur in any order, provided that the requirements of uniprocessor semantics, for example respecting dependencies between instructions within a single processor, are maintained.

A1 \ A2	Normal Read	Device Read		Strongly Ordered Read	Normal Write	Device Write		Strongly Ordered Write
		Non-Shared	Shared			Non-Shared	Shared	
Normal Read				<				<
Device Read (Non-Shared)		<		<		<		<
Device Read (Shared)			<	<			<	<
Strongly Ordered Read	<	<	<	<	<	<	<	<
Normal Write				<				<
Device Write (Non-Shared)		<		<		<		<
Device Write (Shared)			<	<			<	<
Strongly Ordered Write	<	<	<	<	<	<	<	<

Figure B2-1 Memory ordering restrictions

There are no ordering requirements for implicit accesses to any type of memory.

ARM DDI 0100F-02

B2.5.1 Program order for instruction execution

Program order of instruction execution is the order of the instructions in the control flow trace.

Explicit memory accesses in an execution can be either:

Ordered   Denoted by <. Must occur strictly in order.

Weakly Ordered   Denoted by <=. Must occur either in order, or simultaneously.

The rules for determining this for two accesses A1 and A2 are:

If A1 and A2 are generated by two different instructions, then:

*   A1 < A2 if the instruction that generates A1 occurs before the instruction that generates A2 in program order

*   A2 < A1 if the instruction that generates A2 occurs before the instruction that generates A1 in program order.

If A1 and A2 are generated by the same instruction, then:

*   If A1 and A2 are the load and store generated by a SWP or SWPB instruction, then:
    — A1 < A2 if A1 is the load and A2 is the store
    — A2 < A1 if A2 is the load and A1 is the store.

*   If A1 and A2 are two word loads generated by an LDC, LDRD, or LDM instruction, or two word stores generated by an STC, STRD, or STM instruction, excluding LDM or STM instructions whose register list includes the PC, then:
    — A1 <= A2 if the address of A1 is less than the address of A2
    — A2 <= A1 if the address of A2 is less than the address of A1.

*   If A1 and A2 are two word loads generated by an LDM instruction whose register list includes the PC or two word stores generated by an STM instruction whose register list includes the PC, then the program order of the memory operations is not defined.

*   If A1 and A2 are two word loads generated by an LDRD instruction or two word stores generated by an STRD instruction whose register list includes the PC, Rd equals R14 and the instruction is UNPREDICTABLE.

Multiple load and store instructions, such as LDM, LDRD, STM, and STRD, generate multiple word accesses, each of which is a separate access for the purpose of determining ordering.

B2.6 The Weakly Ordered memory model

As described in the introduction to this chapter, the ARM architecture prior to version 6 describes the memory regions rather than the memory order behavior, and as such is open to interpretation and incompatibility across implementations.

ARMv6 is based on the philosophy of a weakly-ordered memory model. In a weakly-ordered model, not all side-effects of a transaction are guaranteed to have completed when the instruction is retired. Behavior is defined for memory-like (Strongly Ordered, Device, and Normal memory) and I/O-like (Strongly Ordered and Device) memory, with a mixture of memory barrier support and software polling required to ensure consistency of system context.

B2.6.1 Memory-like memory support

Actions affecting memory-like memory are observed by the memory agents, that can instigate read or write transactions. A write is observed when the written value is returned by the read of another agent. It is globally observed when the write is returned by a read from any agent in the system. A read is observed until a subsequent write by the observing agent affects the value being read. A read is globally observed until a write by any agent affects the value being read. Where memory has been declared as non-shared, there is only a single agent, meaning that globally observed equates to observation by the local agent.

If an area of memory-like memory is mapped with the Normal or Device attributes, accesses to it must comply with the following:

• the program order model described in *Ordering requirements for memory accesses* on page B2-16
• any constraints specific to the attributes themselves, for example data alignment.

If an area of memory-like memory is mapped as Strongly Ordered, accesses to it must ensure the following:

• all explicit reads and writes to memory-like memory occurring in program order before this instruction are globally observed before the Strongly Ordered access is observed

• a Strongly Ordered access is globally observed before any explicit reads and writes to memory-like memory occurring in program order after this instruction are observed.

B2.6.2 I/O-like memory support

Actions affecting I/O-like memory are observed by the target peripherals when the read or write of the device might begin to affect its state, and trigger any side effects in other devices or normal memory. Reads and writes are globally observed when all target state is updated, and all side effects are complete and visible to the entire system.

If an area of I/O-like memory is mapped with the Device attribute, accesses to it must comply with the following:

• the program order model described in *Ordering requirements for memory accesses* on page B2-16
• any constraints specific to the Device attribute, for example data alignment.

If an area of I/O-like memory is mapped as Strongly Ordered, accesses to it must ensure the following:

- all explicit reads and writes to I/O-like memory occurring in program order before this instruction are observed by the target peripheral(s) before the Strongly Ordered access is observed

- a Strongly Ordered access is observed by the target peripheral(s) before any explicit reads and writes to I/O-like memory occurring in program order after this instruction are observed.

Software polling is required to determine when a Strongly Ordered access is globally observed.

B2.7 Memory Coherency and Access Issues

System designers and programmers need to consider all aspects of a design for overall system correctness. This section outlines some of the problems and pitfalls faced, along with the necessary steps which should be taken to ensure predictable system behavior. It relates the memory model described in this chapter to the ARM architecture provisions of caches, write buffers and memory management (MMU and MPU based) described elsewhere in Part B of this manual.

B2.7.1 Cacheability and bufferability

Because caches and write buffers change the number, type and timing of accesses to main memory, they are not suitable for some types of memory location. In particular, caches rely on normal (memory-like as distinct from I/O-like) memory characteristics such as:

- a load from a memory location returns the last value stored to the location, with no side-effects

- a store to a memory location has no side-effects other than to change the memory location value

- two consecutive loads from a memory location both get the same value

- two consecutive stores to a memory location result in its value becoming the second value stored, and the first value stored is discarded.

Memory-mapped I/O locations usually lack one or more of these characteristics, and so are unsuitable for caching.

Also, write buffers and write-back caches rely on it being possible to delay a store to main memory so that it actually occurs at a later time than the store instruction was executed by the ARM processor. Again, this might not be valid for memory-mapped I/O locations. A typical example is an ARM interrupt handler which stores to an I/O device to acknowledge an interrupt it is generating, and then re-enables interrupts (either explicitly or as a result of the SPSR to CPSR transfer performed on return from the interrupt handler).

If the actual store to the I/O device occurs when the ARM store instruction is executed, the I/O device is no longer requesting an interrupt by the time that interrupts are re-enabled. However, if a write buffer or write-back cache delays the store, the I/O device may still be requesting the interrupt. If so, this results in a spurious extra call to the interrupt handler.

Because of problems like these, both the virtual memory and the protected memory system architectures allow a memory area to be designated as uncacheable, unbufferable or both. From ARMv6, support of memory types is introduced, however, it is IMPLEMENTATION DEFINED whether designating a location as strongly ordered is sufficient to guarantee correctness. Software polling of the I/O device is required to ensure architectural correctness. Details of how the C and B bits are supported for each architecture can be found in TBD_VMSA and TBD_PMSA. Further information on memory mapped I/O can be found in section B2.7.TBDx

The purpose of making a memory area unbufferable is to prevent stores to it being delayed. However, if the area is cacheable and a write-back cache is in use, stores can be delayed anyway. This means that the obvious interpretation of C == 1, B == 0 as cached/unbuffered is not useful for write-back caches. It therefore only has this interpretation in write-through caches. In write-back caches, it instead results in UNPREDICTABLE behavior or selects write-through caching, as shown on page TBD_B2.4.

> **— Note —**
>
> - The reason that a memory-mapped I/O location generally must be marked as uncacheable is effectively to prevent the memory system hardware from incorrectly optimizing away loads and stores to the location. If the I/O system is being programmed in a high-level language, this is not enough. The compiler must be told not to optimize away these loads and stores too. In C and related languages, the way to do this is to use the **volatile** qualifier in the declaration of the memory-mapped I/O location.
>
> - It can also be desirable to mark a memory area as uncacheable for performance reasons. This typically occurs for large arrays which are used frequently, but whose access pattern contains little temporal or spatial locality. Making such arrays uncacheable avoids the cost of loading a whole cache line when only a single access is typically going to occur. It also means that other data items are evicted from the cache less frequently, which increases the effectiveness of the cache on the rest of the data.

B2.7.2   Cache coherency

When a cache and/or a write buffer is used, the system can hold multiple versions of the value of a memory location. Possible physical locations for these values are main memory, the write buffer and the cache. If separate caches are used, either or both of the instruction cache and the data cache can contain a value for the memory location. In a multi-level cache, a cache line may only be present in some levels, having been overwritten or evicted elsewhere.

Not all of these physical locations necessarily contain the value most recently written to the memory location. The memory coherency problem is to ensure that when a memory location is read (either by a data read or an instruction fetch), the value actually obtained is always the value that was most recently written to the location.

In the ARM memory system architectures, some aspects of memory system coherency are required to be provided automatically by the system. Other aspects are dealt with by memory coherency rules, which are limitations on how programs must behave if memory coherency is to be maintained. The behavior of a program that breaks a memory coherency rule is UNPREDICTABLE. Address mapping and caches require careful management to ensure memory coherency at all times. Cache and write buffer management typically requires a sequence containing one or more of the following:

- cleaning the data cache if it is a write-back cache
- invalidating the data cache
- invalidating the instruction cache
- draining the write buffer
- performing a prefetch flush on the instruction pipeline.

Prior to ARMv6, the operations and sequences are IMPLEMENTATION DEFINED. In ARMv6, the memory order model, cache, TLB and memory barrier operations supported in the System Control Coprocessor (CP15) allow the operating system support to be standardized for level 1 memory.

Address mapping changes

In an ARM memory system that implements virtual-to-physical address mapping (such as the MMU-based memory system described in Chapter B4 *Virtual Memory System Architecture*), there are two implementation choices for the address (TAG) associated with a cache line:

- it can be the virtual address of the data in the cache line
- it can be the physical address of the data in the cache line.

If an implementation is designed to use the virtual address, a change to the virtual-to-physical address mapping can cause major memory coherency problems, as any data in the remapped address range which is in the cache ceases to be associated with the correct physical memory location.

Similarly, the data in a write buffer can have virtual or physical addresses associated with it, depending on whether the address mapping is done when data is placed in the write buffer or when it is stored from the write buffer to main memory. If a write buffer is designed to use the virtual address, a change to the virtual-to-physical address mapping can again cause memory coherency problems.

These problems can be avoided by performing a sequence of cache and/or write buffer operations before a change of virtual-to-physical address mapping. There may also be requirements for the code that performs the change of address mapping and any data it accesses to be uncacheable, unbufferable or both. As part of the address mapping change sequence, there might also be a need to invalidate translation buffer entries associated with an MMU.

From ARMv6, defined System Control Coprocessor (CP15) operations are mandated for level 1 cache management and translation buffer control.

──── **Note** ────

Implementors are strongly advised to work with ARM where control of additional cache levels is required, to minimize potential impacts of future compatibility.

Instruction cache coherency

A memory system is permitted to satisfy an instruction fetch request from a separate instruction cache. An instruction cache line fetch can be satisfied from main memory, and there is no requirement for data stores to update a separate instruction cache. This means that the following sequence of events causes a potential memory coherency problem:

1. An instruction is fetched from an address A1, causing the cache line containing that address to be loaded into the instruction cache.

2. A data store occurs to an address A2 in the same cache line as A1, causing an update to one or more of the data cache, the write buffer, and main memory, but not to the instruction cache. (A2 might be the same address as A1, or a different address in the same cache line. The same considerations apply in both cases.)

3. An instruction is executed from the address A2. This could result in either the old contents or the new contents of the memory location being executed, depending on whether the cache line is still present in the instruction cache or must be reloaded.

This problem can be avoided by performing a sequence of cache control operations between steps 2 and 3. Typically, this sequence consists of:

• nothing at all for an implementation with a unified cache

• invalidating the instruction cache for an implementation with separate caches and a write-through data cache

• cleaning the data cache followed by invalidating the instruction cache for an implementation with separate caches and a write-back data cache.

Therefore, the memory coherency rule to maintain instruction cache coherency is that: if a data store writes an instruction to cacheable memory, a sequence of instructions to flush any possible cache entries must be executed before the instruction is executed. A typical case where this must be done is when an executable file is loaded into memory. After loading the file and before branching to the entry point of the newly loaded code, a code sequence must be executed to ensure that the newly loaded program executes correctly.

The performance cost of the cache cleaning and invalidating required when this happens can be large, both as a direct result of executing the cache control operations and indirectly because the instruction cache must be reloaded. This means that programming techniques that involve frequent real-time generation and execution of small code fragments must be treated with care, as porting them to a new memory system can require careful redesign and re-optimization to avoid excessive performance loss.

———— **Note** ————

• The sequence required to maintain instruction cache coherency is part of the sequence executed by an Instruction Memory Barrier, but not necessarily all of it. See section TBD_B2.8 for more details.

• On some implementations, it is possible to exploit knowledge of the range of addresses occupied by newly stored instructions to reduce the cost of the required cache operations. For example, it might be possible to restrict the cache cleaning and invalidating to that address range.

• If it is known that none of the range of addresses containing newly stored instructions is in the instruction cache, the memory coherency problem described above cannot occur. However, it is difficult to be certain of this across all ARM implementations because:

— A fetch of any instruction in a cache line causes all of the instructions in that cache line to be loaded into the instruction cache.

— Typically, some instructions are fetched but never executed, so it is possible for an instruction cache line to have been loaded but not to contain any executed instructions. Also, although instructions that are fetched but not executed are typically close to instructions that have been executed, this need not be the case in implementations that use branch prediction or similar techniques.

As a result, code that uses this technique to avoid the instruction cache coherency problem is not fully implementation-independent.

Prefetching and self-modifying code

Many ARM implementations fetch instructions from memory before execution of the instructions that precede them has completed. This behavior is known as *prefetching* the instruction.

Prefetching an instruction does not commit the ARM implementation to actually executing the instruction. Two typical cases in which the instruction is not subsequently executed are:

• When an exception occurs, execution of the current instruction is completed, all further prefetched instructions are discarded, and execution of the instructions at the exception vector is started.

• When a branch is taken, or a predicted branch is not taken, any instructions that have already been prefetched are discarded.

ARM implementations are free to choose how far ahead of the current point of execution they prefetch instructions, or even to have a dynamically varying number of prefetched instructions. The original ARM implementation prefetched two instructions ahead of the instruction currently being executed, but implementations are free to choose to prefetch more or less than this.

When an instruction reads the PC, it gets the address of the instruction that is two after itself:
• for ARM instructions, it gets its own address plus 8
• for Thumb instructions, it gets its own address plus 4.

Historically, there is a link between this two-instruction offset for PC reads and the two-instruction prefetch of the original ARM implementation. However, this link is not architectural. An implementation which prefetches a different number of instructions still ensures that an instruction which reads the PC gets the address of the instruction two after itself.

As well as being free to choose how many instructions to prefetch, an ARM implementation can choose which possible future execution path to prefetch along. For example, after a branch instruction, it can choose to prefetch either the instruction following the branch or the instruction at the branch target. This is known as branch prediction.

A potential problem with all forms of instruction prefetching is that the instruction in memory might be changed after it was prefetched but before it is executed. If this happens, the modification to the instruction in memory does not normally prevent the already prefetched copy of the instruction from executing to completion.

For example, in the following code sequence, the STR instruction replaces the SUB instruction that follows it by a copy of the ADD instruction:

```
    LDR     r0, AddInstr
    STR     r0, NextInstr
NextInstr
    SUB     r1, r1, #1
    :
    :
AddInstr
    ADD     r1, r1, #1
```

However, the instruction executed after the STR instruction is normally the SUB instruction on the first occasion that this code is executed, because the SUB instruction was prefetched before the instruction in memory was changed. The ADD instruction is not executed until the second time that the code sequence is executed.

Furthermore, even this behavior cannot be guaranteed, because:

• On the first occasion that the code sequence is executed, it is possible that an interrupt will occur immediately after the STR. If it does, the SUB instruction that had been prefetched is discarded. When the interrupt handler returns, the instruction at NextInstr is prefetched again, and is the ADD instruction this time. So while the SUB instruction is normally executed on the first occasion that the code is executed, it is possible that the ADD instruction is executed instead.

• Either the ARM processor or the memory system is allowed to keep copies of instructions fetched from memory and use those copies instead of repeating the instruction fetch if the instruction is executed again. If this occurs, a copy of the SUB instruction can be executed on the second or subsequent occasion that the code sequence is executed.

The main reason that this might occur is that the memory system contains separate instruction and data caches. However, other possibilities also exist. For example, some forms of branch prediction hardware keep copies of the instructions at branch targets.

The overall result is that code which writes one or more instructions to memory and then executes them (known as self-modifying code) cannot be executed reliably on ARM processors without special precautions, which include cache management code sequences similar to those already described for instruction cache coherency. Programming techniques that involve the use of self-modifying code are to be avoided as far as possible.

B2.7.3 Direct Memory Access (DMA) operations

I/O devices can perform Direct Memory Access (DMA) operations, in which they access main memory directly, without the processor performing any accesses to the data concerned.

If a DMA operation stores to main memory without updating the cache and/or write buffer, some rules normally relied upon to simplify memory coherency issues might be violated. For example, it is normally the case that if a data item is in the cache, the copy of it in main memory is not newer than the copy in the cache. This allows the value in the cache to be returned for a data load without explicitly checking whether there is a more recently written version in main memory. However, a DMA store to main memory can cause the main memory value to be more recently written than the cache value.

Similarly, if a DMA operation loads data from main memory without also checking the cache and/or write buffer to see whether they contain more recent versions, it might get an out-of-date version of the data.

In both cases, a possible solution would be for DMA to also access the cache and write buffer. However, this would significantly complicate the memory system.

So, a memory system implementation can have IMPLEMENTATION DEFINED memory coherency rules for handling DMA operations.

Typically, these involve one or more of the following:

- making the memory areas involved in the DMA operation as uncacheable and/or unbufferable

- cleaning and/or invalidating the data cache, at least with respect to the address range involved in the DMA operation

- draining the write buffer

- restrictions on processor accesses to the address range involved in the DMA operation until it is known that the DMA operation is complete.

B2.7.4 Memory-mapped I/O (I/O-like memory)

The standard way to perform I/O functions on ARM systems is by the use of memory-mapped I/O. This uses special memory addresses which supply I/O functions when they are loaded from or stored to. Typically, loading from a memory-mapped I/O address is used for input, and storing to a memory-mapped I/O address is used for output. Both loads and stores can also be used to perform control functions, either instead of or in addition to their normal input or output function.

The behavior of a memory-mapped I/O location usually differs from that expected of a normal memory location. For example, two successive loads from a normal memory location return the same value each time unless there has been an intervening store to that location. For a memory-mapped I/O location, the value returned by the second load can be different from the value returned by the first load. Typically, this is because the first load has a side-effect (such as removing the loaded value from a buffer) or because of a side-effect of an intervening load or store to another memory-mapped I/O location.

These differences mean memory-mapped I/O locations are normally marked as uncacheable and unbufferable (or from ARMv6, marked as strongly ordered or device memory type), to avoid changes to the number, type, order, or timing of the accesses made to them.

—— **Note** ——

In ARMv6, memory mapped I/O can be classified as device memory where the access order only needs to be restricted to accesses within this memory type (or a particular device). Strongly ordered memory applies stricter order rules on the accesses, the order rule applying to all memory types; strongly ordered, device and normal memory.

Instruction fetches from memory-mapped I/O

ARM implementations can vary considerably with regard to when they fetch instructions from memory. As a result, it is strongly recommended that memory-mapped I/O locations are only used for data loads and stores, not for instruction fetches. From ARMv6, instruction fetches shall be from memory-like memory, otherwise they are UNPREDICTABLE.

Data accesses to memory-mapped I/O

An instruction sequence accesses data memory at various points during its execution, generating a sequence of load and store accesses. Provided these loads and stores access normal memory locations, they only interact with each other if they access the same memory location. As a result, loads and stores to distinct normal memory locations can be performed in a different order to that implied by the instruction sequence, without changing the final result of the sequence. This freedom to change the order of memory accesses can be exploited by a memory system to improve performance (for example, by the use of caches and write buffers).

Furthermore, data accesses to the same normal memory location have other properties that can be exploited to improve performance. These include:

*   Successive loads from the same location without an intervening store generate identical results.

*   A load from a location returns the last value stored to that location.

*   Multiple accesses of one data size can sometimes be merged into a single, larger size access. For example, separate stores to the two halfwords contained within a word can be merged to produce a single word store.

However, if the memory words, halfwords or bytes accessed by the code sequence are memory-mapped I/O locations, one access can generate a side-effect which changes the results of a subsequent access to a different location. If this happens, the time order of individual accesses makes a difference to the final results of the code sequence. Also, a load access to a memory-mapped I/O location can have a side-effect that changes the result of a subsequent access to the same location. Accesses to memory-mapped I/O locations must therefore not be optimized away, and their time order must not be changed.

It is also important that for memory-mapped I/O, the data size of each memory access is maintained. For example, a code sequence that specifies 4 byte reads from 4 sequential byte addresses must not be merged into a single word read when accessing memory-mapped I/O. Such a system might cause the final results of the code sequence to be different from that intended. Similarly a system which splits word accesses up into many byte accesses might cause memory-mapped I/O devices not to operate as expected.

Each ARM implementation provides a mechanism to ensure that no changes are made to the number of accesses in a sequence of data memory accesses, or to their data sizes, or time order. This mechanism consists of requirements on the memory accesses whose number, data sizes, and time order are to be preserved. If these requirements are not adhered to for accesses to memory-mapped I/O locations, unexpected behavior might occur. The requirements are IMPLEMENTATION DEFINED prior to ARMv6.

 Typical requirements include:

*   Constraints on memory attributes of the memory-mapped I/O locations. For example, in the standard memory system architectures the memory locations must be uncacheable and unbufferable, and abide by the memory type rules from ARMv6.

*   Constraints on the sizes or alignments of the accesses to the memory-mapped I/O locations. For example, if an ARM implementation has a 16-bit external data bus, it might prohibit the use of 32-bit accesses to memory-mapped I/O locations, since they cannot be performed in a single bus cycle.

- A requirement for additional external hardware. For example, an alternative possibility for an ARM implementation with a 16-bit external bus is to allow 32-bit accesses to memory-mapped I/O locations, but require external hardware to re-assemble the two 16-bit bus accesses into a single 32-bit access to the I/O device.

If a sequence of data memory accesses includes some accesses which meet the requirements for memory-mapped I/O accesses and some which do not, then:

- The number and data sizes of the accesses that meet the requirements are preserved. In particular, they are not merged with each other or with the accesses that do not meet the requirements in any way. The accesses which do not meet the requirements can be merged with each other.

- The time order of the accesses which meet the requirements are preserved relative to each other. Their time order relative to accesses which do not meet the requirements is not guaranteed.

Time ordering of LDM and STM instructions

The LDM instruction performs a sequence of loads from successive words in memory, and the STM instruction performs a similar sequence of stores. The rules described above for accessing memory-mapped I/O apply to the sequence of word accesses within one of these instructions in the same way as they do to a series of separate memory access instructions.

The time order of the sequence of memory accesses performed by an LDM or STM instruction is only architecturally defined under limited circumstances. The rules for this are:

- If the register list in the instruction includes the PC, the time order of the sequence of memory accesses is not defined. (This means that such LDM and STM instructions are not suitable for accessing memory-mapped I/O.)

- If the register list in the instruction does not include the PC, the time order of the sequence of memory accesses is in order of memory address, starting with the lowest address and ending with the highest address. (This order is identical to ascending register number order within the list of registers to be loaded or stored.)

- If all of the memory accesses generated by an LDM or STM meet the IMPLEMENTATION DEFINED requirements to be treated as memory-mapped I/O locations, then their number, data sizes, and time order are preserved.

If some of the memory accesses generated by an LDM or STM meet the IMPLEMENTATION DEFINED requirements to be treated as memory-mapped I/O locations, but others do not, then their number, data sizes, and time order are not guaranteed to be preserved. In particular, the ARM processor and memory system do not even necessarily preserve the relative time order of the accesses that do meet the requirements. This is an exception to the normal rules that govern what happens when some accesses meet the requirements and others do not.

For example, with the standard memory systems the time order of the memory accesses is not guaranteed to be preserved if the LDM or STM crosses the boundary between a cacheable area of memory and an uncacheable, unbufferable area. Such LDM and STM instructions are therefore not suitable for memory-mapped I/O.

B2.8   Memory Barriers

Memory barrier is the general term applied to an instruction, or sequence of instructions, used to force synchronization events by a processor with respect to retiring load/store instructions in a processor core. A memory barrier is used to guarantee completion of preceding load/store instructions to the programmers model, flushing of any prefetched instructions prior to the event, or both. ARMv6 mandates three explicit barrier instructions in the System Control Coprocessor to support the memory order model described in *The Weakly Ordered memory model* on page B2-18, and requires these instructions to be available in both privileged and user modes:

- DataMemoryBarrier as described in *DataMemoryBarrier (DMB) CP15 Register 7* on page B2-32
- DataWriteBarrier as described in *DataWriteBarrier (DWB) CP15 Register 7* on page B2-32
- PrefetchFlush as described in section *PrefetchFlush CP15 Register 7* on page B2-33.

These instructions may be sufficient on their own, or may need to be used in conjunction with cache and memory management maintenance operations; operations which are only available in privileged modes. Support of memory barriers in earlier versions of the architecture is IMPLEMENTATION DEFINED.

B2.8.1   Instruction Memory Barriers (IMBs)

While self-modifying code in general is considered undesirable, it is impossible to avoid its use entirely. For example, any system which allows a program to be loaded into memory and then executed is using self-modifying code. It is therefore necessary to define a sequence of operations that can be used in the middle of a self-modifying code sequence to make it execute reliably. This sequence is called an *Instruction Memory Barrier* (IMB), and may depend both on the ARM processor implementation and on the memory system implementation.

The IMB sequence must be executed after the new instructions have been stored to memory and before they are executed, for example, after a program has been loaded and before its entry point is branched to. Any self-modifying code sequence which does not use an IMB in this way has UNPREDICTABLE behavior.

An IMB may be included in-line where needed, however, it is recommended that software is designed so that the IMB sequence is provided as a call to an easily replaceable system dependencies module. This eases porting across different architecture variants, ARM processors, and memory systems.

IMB sequences may include operations that are only usable from privileged processor modes, such as the cache cleaning and invalidation operations supplied by the standard System Control coprocessor. To allow User mode programs access to privileged IMB sequences, it is recommended that they are supplied as operating system calls, invoked by `SWI` instructions. For systems that use the 24-bit immediate in a `SWI` instruction to specify the required operating system service, ARM has defined default values as follows:

```
SWI 0xF00000; the general case
SWI 0xF00001; where the system can take advantage of specifying an
           ; affected address range
```

These are recommended for general use unless an operating system has good reason to choose differently, to align with a broader range of operating system specific system services.

The `SWI 0xF00000` call takes no parameters, does not return a result, and should use the same calling conventions as a call to a C function with prototype:

```
        void IMB(void);
```

apart from the fact that a `SWI` instruction is used for the call, rather than a `BL` instruction.

The `SWI 0xF00001` call should use similar calling conventions to those used by a call to a C function with prototype:

```
 void IMB_Range(unsigned long start_addr, unsigned long end_addr);
```

where the address range runs from `start_addr` (inclusive) to `end_addr` (exclusive). When the standard ARM Procedure Call Standard is used, this means that `start_addr` is passed in R0 and `end_addr` in R1.

On some ARM implementations, the execution time cost of an IMB can be very large (many thousands of clock cycles), even when a small address range is specified. For small scale uses of self-modifying code, this is likely to lead to a major loss of performance. It is therefore recommended that self-modifying code is only used where it is unavoidable and/or it produces sufficiently large execution time benefits to offset the cost of the IMB.

Other uses for IMBs

Some memory systems allow virtual-to-physical address mapping, in which the physical memory location corresponding to an address generated by the ARM processor can be changed. If this address mapping is changed after an instruction has been prefetched but before it is executed, and the address of the instruction is affected by the change of address mapping, then the wrong instruction is executed.

This is very similar to the situation that arises if a store occurs to an instruction address after it has been prefetched but before it is executed. In both cases, the instruction held at the memory address is being changed, either because a value is being stored to it or because a different physical memory location becomes associated with the address. The same solution is therefore used when the virtual-to-physical address mapping is changed. The IMB sequence must be executed after a change of virtual-to-physical address mapping and before any attempt to execute an instruction from a memory area whose address mapping has been changed.

Another similar case occurs if memory access permissions are changed between prefetching and executing an instruction. If access was not permitted when the instruction was prefetched but is permitted when it is executed, an unexpected Prefetch Abort exception might occur. In the opposite case that access was permitted when the instruction was prefetched and is no longer permitted when it is executed, there might be a security hole in the system.

Memory access permissions can typically be changed either by explicitly writing new access permission settings to the memory system, or because the memory system supports different access permissions for User mode and privileged modes and one of the following occurs:

- an exception occurs in User mode, causing the processor to switch to a privileged mode

- privileged code changes mode to User mode.

All ARM implementations ensure that the following events do not cause any instructions to be executed after having been prefetched with the wrong access permissions:

• An exception occurring in User mode.

• Execution of one of the instructions designed for exception return causing a change from a privileged mode to User mode. These instructions are the ones which have a side-effect of copying the SPSR of the current mode to the CPSR, namely:

— The data processing instructions ADCS, ADDS, ANDS, BICS, EORS, MOVS, MVNS, ORRS, RSBS, RSCS, SBCS and SUBS when their destination register is R15. (However, only MOVS and SUBS are commonly used for exception return.)

— The form of the LDM instruction described in TBD.

The same is not guaranteed in the remaining cases where memory access permissions might change between prefetching and executing an instruction. These are:

• explicitly writing new access permission settings to the memory system

• changing from a privileged mode to User mode by means of an MSR instruction.

In these cases, an IMB sequence needs to be executed shortly after the change of access permissions, and none of the instructions executed after the change of access permissions and before the Instruction Memory Barrier should be affected by the change of access permissions.

However, the cost of a full IMB can often be avoided in these cases. In particular, the instruction word associated with any particular address has not changed, so it is usually possible to avoid cache flushes. An implementation can therefore define restricted versions of the IMB sequence to be used in these cases.

In the case of an MSR instruction changing from a privileged mode to User mode, a restricted version of the IMB sequence that works on all ARM processors to date is simply to execute any instruction that writes to the PC, other than the branch instructions described in the following sections:

• TBD Crossrefelemtextefault   Fonton page

• Crossrefelemtextefault   Fonton page

• Crossrefelemtextefault   Fonton page

• Crossrefelemtextefault   Fonton page

• Crossrefelemtextefault   Fonton page.

In other words, the mode change should not affect the access permissions of any instructions that can be reached from the MSR instruction by any combination of:

• Normal sequential execution of instructions.

• For each branch from the above list that can be reached in this way, execution of the instruction at its target. (The branch instructions in the list are precisely those that have a fixed, statically determined target.)

This set of instructions is sometimes referred to as the set of instructions that can be reached by predictable subsequent execution from the MSR instruction.

B2.9 Explicit Memory Barriers

Two explicit Memory Barrier instructions, *DataMemoryBarrier* (DMB) and *DataWriteBarrier* (DWB), are described in this section. In addition, to ensure correct operation where a processor writes code (self modifying code, or code that is copied to memory prior to execution), an explicit pipeline flush (PrefetchFlush) instruction is provided.

All implementations supporting the weak order model defined in ARMv6 must support all three coprocessor instructions.

Explicit memory barriers affect reads and writes to the memory system generated by load and store instructions being executed in the CPU. Reads and writes generated by L1 DMA transactions, and instruction fetches or accesses caused by a hardware page table access, are not explicit accesses.

B2.9.1 DataMemoryBarrier (DMB) CP15 Register 7

DMB acts as a general memory barrier, exhibiting the following behavior:

- All explicit reads and writes to memory-like memory occurring in program order before this instruction are globally observed before any explicit memory reads and writes to memory-like memory after this instruction are observed.

- Reads and writes to I/O-like memory (with the Strongly Ordered or Device memory attribute) occurring in program order before the DMB might not be globally observed before this instruction completes. Side effects might still be in progress. As a result, software polling of system (peripheral) status registers is required to ensure completion of all side effects.

- No explicit memory transactions occurring in program order after this instruction are started until this instruction completes.

- Other instructions occurring in program order before the DMB can complete out of order with the DMB.

The encoding for DataMemoryBarrier is described in section TBD???

B2.9.2 DataWriteBarrier (DWB) CP15 Register 7

DWB acts as a special kind of memory barrier, exhibiting the following behavior:

- DWB completes when all explicit write transactions to memory-like memory occurring in program order before this instruction are globally observed. All explicit read transactions to memory-like memory occurring in program order before the DWB might not be globally observed before this instruction completes.

- Reads and writes to I/O-like memory (with the Strongly Ordered or Device memory attribute) occurring in program order before the DWB might not be globally observed before this instruction completes. Side effects might still be in progress. As a result, software polling of system (peripheral) status registers is required to ensure completion of all side effects.

- No explicit memory transactions occurring in program order after this instruction is started until this instruction completes.

- No instructions occurring in program order after the DWB can complete until this instruction completes.

This instruction is synonymous with DrainWriteBuffer in earlier versions of the architecture. DrainWriteBuffer is a historical legacy, the name change more accurately reflecting the updated behavior.

The encoding for DataWriteBarrier is described in TBD section ???

B2.9.3 PrefetchFlush CP15 Register 7

The PrefetchFlush instruction flushes the pipeline in the processor, so that all instructions following the pipeline flush are fetched from memory, including the cache, after the instruction has been completed. Combined with DataWriteBarrier, and potentially invalidating the instruction cache, this ensures that any instructions written by the processor before the PrefetchFlush are executed. Such a guarantee is required as part of the mechanism for handling self-modifying code. The execution of a DataWriteBarrier instruction and the invalidation of the Instruction cache and branch target cache are also required for the handling of self-modifying code.

The PrefetchFlush is guaranteed to perform this function, and alternative methods of performing the same task, such as a branch instruction, might be optimized in the hardware to avoid the pipeline flush (for example, by the use of a branch predictor).

The encoding for the PrefetchFlush is described in section TBD ???

Chapter B3
The System Control Coprocessor

This chapter describes coprocessor 15, the System Control coprocessor. It contains the following sections:

B3.1   About the System Control coprocessor

All of the standard memory and system facilities are controlled by coprocessor 15 (CP15), which is known as the System Control coprocessor. Some also use other methods of control, which are described in the chapters describing the facilities concerned. For example, the Memory Management Unit described in Chapter B4 *Virtual Memory System Architecture* is also controlled by page tables in memory.

ARMv6 systems shall include a System Control Coprocessor, with support for automatic interrogation of cache, tightly coupled memory, and coprocessor provision. It also provides the control mechanism for memory management (MMU and MPU support as applicable).

Prior to ARMv6, CP15 instructions are UNDEFINED when CP15 is not implemented, however, CP15 has become a defacto standard for processor ID, cache control, and memory management (MMU and MPU support) in implementations since ARMv4. This manual should be read in conjunction with the relevant implementation reference manual to determine the exact details of CP15 support in a particular part.

This chapter describes the overall design of the System Control coprocessor and how its registers are accessed. Detailed information is given on some of its registers. Other registers are allocated to facilities described in detail in other chapters and are only summarized in this chapter.

B3.2 Registers

The System Control coprocessor can contain up to 16 primary registers, each of which is 32 bits long. Additional fields in the register access instructions are used to further refine the access, increasing the number of physical 32-bit registers in CP15. The 4-bit primary register number is used to identify registers in descriptions of the System Control coprocessor, because it is the primary factor determining the function of the register.

CP15 registers can be read-only, write-only or read/write. The detailed descriptions of the registers specify:

- what types of access are allowed

- what functionality is invoked by each type of access

- whether a primary register identifies more than one physical register, and if so, how they are distinguished

- any other details that are relevant to the use of the register.

B3.2.1 Register access instructions

The only defined System Control coprocessor instructions are:
- MCR instructions to write an ARM register to a CP15 register
- MRC instructions to read the value of a CP15 register into an ARM register
- MCRR instructions for range operations introduced in ARMv6, and optional in earlier versions of the architecture.

All CP15 CDP, LDC and STC instructions are UNDEFINED.

The format of the MCR/MRC instructions are illustrated below, with bits[11:8](*cp_num*) indicating CP15, and the CRn field indicating the primary register number, with CRm and opcode2 providing additional register decode.

31 28	27 26 25 24	23 21	20	19 16	15 12	11 8	7 5	4	3 0
cond	1 1 1 0	opcode1	L	CRn	Rd	1 1 1 1	opcode2	1	CRm
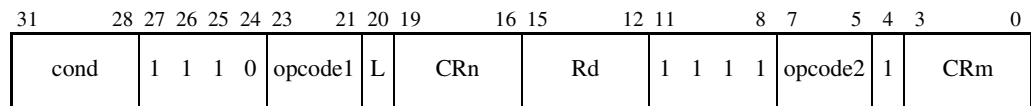
The MCR and MRC instructions to access the CP15 registers use the generic syntax for those instructions:

```
MCR{<cond>}  p15, 0, <Rd>, <CRn>, <CRm>{, <opcode2>}    (L = 0)
MRC{<cond>}  p15, 0, <Rd>, <CRn>, <CRm>{, <opcode2>}    (L = 1)
```

where:

<cond>   This is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-7. If <cond> is omitted, the AL (always) condition is used.

Bits[23:21]   These bits of the instruction, which are the `<opcode1>` field in generic `MRC` and `MCR` instructions, are generally 0b000 in valid CP15 instructions. However, `<opcode1> == 1` is being used for level 2 cache support and considered for some other specialist tasks. Unassigned values are UNPREDICTABLE.

`<Rd>`   This is the ARM register involved in the transfer (the source register for `MCR` and the destination register for `MRC`). This register must not be R15, even though `MCR` instructions normally allow it to be R15. If R15 is specified for `<Rd>` in a CP15 `MRC` or `MCR` instruction, the instruction is UNPREDICTABLE.

`<CRn>`   This is the primary CP15 register involved in the transfer (the destination register for `MCR` and the source register for `MRC`). The standard generic coprocessor register names are c0, c1, ..., c15.

`<CRm>`   This is an additional coprocessor register name which is used for accesses to some primary registers to specify additional information about the version of the register and/or the type of access.

When the description of a primary register does not specify `<CRm>`, c0 must be specified. If another register is specified, the instruction is UNPREDICTABLE.

`<opcode2>`   This is an optional 3-bit number which is used for accesses to some primary registers to specify additional information about the version of the register and/or the type of access. If it is omitted, 0 is used.

When the description of a primary register does not specify `<opcode2>`, it must be omitted or 0 must be specified. If another value is specified, the instruction is UNPREDICTABLE.

The MCRR format (see *MCRR* on page A4-63) has less scope for decode. The primary register is implied (no CRn field), and the CRm and opcode fields are used to decode the correct function.

Prior to ARMv6, `MCR` and `MRC` instructions can only be used when the processor is in a privileged mode. If they are executed when the processor is in User mode, an Undefined Instruction exception occurs

ARMv6 introduced user access of the following commands:
- Prefetch flush
- Data write barrier
- Data memory barrier
- Clean and prefetch range operations.

──── **Note** ────

If access to privileged System Control coprocessor functionality by User mode programs is required, the usual solution is that the operating system defines one or more SWIs to supply it. As the precise set of memory and system facilities available on different processors can vary considerably, it is recommended that all such SWIs are implemented in an easily replaceable module and that the SWI interface of this module is defined to be as independent of processor details as possible.

The IMB and IMB_Range SWIs described in *Instruction Memory Barriers (IMBs)* on page B2-29 are examples of such SWIs.

B3.2.2   Primary register allocation

Table B3-1 shows the allocation of the primary registers of the System Control coprocessor.

Table B3-1 Primary register allocation

Reg	Generic use	Specific uses	Details in
0	ID codes (read-only)	Processor ID, Cache, Tightly-coupled Memory and TLB type	*Register 0: ID codes* on page B3-7
1	Control bits (read/write)	System Configuration Bits	TBD
2	Memory protection and control	Page Table Control	TBD
3	Memory protection and control	Domain Access Control	TBD
4	Memory protection and control	Reserved	TBD
5	Memory protection and control	Fault status	TBD
6	Memory protection and control	Fault address	TBD
7	Cache and write buffer	Cache/write buffer control	TBD
8	Memory protection and control	TLB control	TBD
9	Cache and write buffer	Cache lockdown	TBD
10	Memory protection and control	TLB lockdown	TBD
11	Tightly-coupled Memory Control	DMA Control	-
12	Reserved	-	-
13	Process ID	Process ID	*Register 13: Process ID*. See TBD xref.
14	Reserved	-	-
15	IMPLEMENTATION DEFINED	IMPLEMENTATION DEFINED	Implementation documents

B3.3 Register 0: ID codes

CP15 register 0 contains one or more identification codes for the ARM and system implementation. When this register is read, the opcode2 field of the MRC instruction selects which identification code is wanted, as shown in Table B3-2, and the CRm field must be specified as c0 (if it is not, the instruction is UNPREDICTABLE). Writing to CP15 register 0 is UNPREDICTABLE.

Table B3-2 System Control coprocessor ID registers

opcode2	Register	Details in
0b000	Main ID register	*Main ID register*
0b001	Cache Type register	*Cache Type register* on page B3-11
0b010	*Tightly Coupled Memory* (TCM) Type register	*TCM Type register* on page B3-11
0b011	TLB Type register	
0b100	MPU type register (PMSAv6)	
other	Reserved (see main text)	-

If an <opcode2> value corresponding to an unimplemented or reserved ID register is encountered, the System Control coprocessor returns the value of the main ID register.

ID registers other than the main ID register are defined so that when implemented, their value cannot be equal to that of the main ID register. Software can therefore determine whether they exist by reading both the main ID register and the desired register and comparing their values. If the two values are not equal, the desired register exists.

B3.3.1 Main ID register

When CP15 register 0 is read with <opcode2> == 0, an identification code is returned from which, among other things, the ARM architecture version number can be determined, as well as whether or not the Thumb instruction set has been implemented.

―――― **Note** ――――

Only some of the fields in CP15 register 0 are architecturally defined. The rest are IMPLEMENTATION DEFINED and provide more detailed information about the exact processor variant. Consult individual datasheets for the precise identification codes used for each processor.

―――――――――――――

Implementor code

Bits[31:24] of the main ID register contain an implementor code.

The following codes are defined (all other values of the architecture code are reserved by ARM Limited.):

0x41   A (ARM Limited)

`0x44`	D (Digital Equipment Corporation)
`0x49`	M (Motorola)
`0x69`	i (Intel Corporation)

ARM processor implementation IDs

For historical reasons, there are a variety of ways in which the CP15 register 0 ID code might need to be interpreted. If bit[19] is zero, bits[15:12] should be interpreted as follows:

*   if they are `0x0`, this indicates an OBSOLETE part (pre-ARMv4 architecture)
*   if they are `0x7`, this indicates that the processor is in the ARM7 family
*   if > `0x7`, a more recent processor family than ARM7 is involved.

ARM7 processor IDs are interpreted as follows:

31 24	23	22 16	15 4	3 0
Implementor	A	Variant	Primary part number	Revision

Bits[3:0]   Contain the IMPLEMENTATION DEFINED revision number for the processor.

Bits[15:4]   Contain the IMPLEMENTATION DEFINED representation of the primary part number for the processor. The top four bits of this number are `0x7`.

Bits[22:16]   Contain an IMPLEMENTATION DEFINED variant number.

Bit[23]   Indicates which of the two possible architectures for an ARM7-based process is involved:

  0   Architecture 3 (OBSOLETE part)

1   Architecture 4T.

Bits[31:24]   `0x41` = A (ARM Limited) implementation code.

Processor implementations since ARM7 have a general format of bits[23:0] which are common across implementations from ARM and architecture licensees. Two general formats are defined, dependent on the value of bit[19]. They are described in the following sections.

Post-ARM7 processors (General format 1: bit[19]==0)

If bits[15:12] of the ID code are neither `0x0` nor `0x7`, the ID code is interpreted as follows:

31 24	23 20	19 16	15 4	3 0
Implementor	Variant	Architecture	Primary part number	Revision

Bits[3:0]   Contain the IMPLEMENTATION DEFINED revision number for the processor.

Bits[15:4] Contain an IMPLEMENTATION DEFINED representation of the primary part number for the processor. The top four bits of this number are not allowed to be `0x0` or `0x7`.

Bits[19:16] Contain an architecture code. The following architecture codes are defined:

`0x1`	ARM architecture v4
`0x2`	ARM architecture v4T
`0x3`	ARM architecture v5
`0x4`	ARM architecture v5T
`0x5`	ARM architecture v5TE
`0x6`	ARM architecture v5TEJ
`0x7`	ARM architecture v6.

All other values of the architecture code are reserved by ARM Limited

Bits[23:20] Contain an IMPLEMENTATION DEFINED variant number. This is typically used to distinguish two variants of the same primary part, for example, two different cache size variants.

Bits[31:24] Contain an implementor code. See *Implementor code* on page B3-7.

Post-ARM7 processors (General format 2: bit[19] ==1)

TBD.

ARM DDI 0100F-02

B3.3.2   Cache Type register

The Cache Type register supplies the following details about the cache:

*   whether it is a unified cache or separate instruction and data caches
*   its size, line length and associativity
*   whether it is a write-through cache or a write-back cache
*   cache cleaning and lockdown capabilities

The format of the Cache Type register is:

31	29 28	25 24 23	12 11	0

0 0 0	ctype	S	Dsize	Isize

ctype   Specifies details of the cache not specified by the S bit and the Dsize and Isize fields. All values not specified in the table are reserved for future expansion.

S bit   Specifies whether the cache is a unified cache (S == 0), or separate instruction and data caches (S == 1). If S == 0, the Isize and Dsize fields both describe the unified cache, and must be identical.

Dsize   Specifies the size, line length and associativity of the data cache, or of the unified cache if S == 0.

Isize   Specifies the size, line length and associativity of the instruction cache, or of the unified cache if S == 0.

A detailed discussion on caches is provided in Chapter B6 *Caches and Write Buffers*. See *Cache Type register* on page B6-13 for the encoding of the cache type register fields.

B3.3.3   TCM Type register

The format of the Tightly-Coupled Memory (TCM) Type register is:

31	29 28	19 18	16 15	3 2	0

0 0 0	SBZ/UNP	DTCM	SBZ/UNP	ITCM

ITCM (Bits[2:0]) Indicate the number of Instruction (or Unified) Tightly-Coupled Memories implemented. This value lies in the range 0-4, all other values are reserved. All Instruction TCMs must be accessible to both instruction and data sides.

DTCM (Bits[18:16]) Indicate the number of Data Tightly-Coupled Memories implemented. This value lies in the range 0-4, all other values are reserved.

A detailed discussion of tightly vcoupled memory is provided in chapter Chapter B7 *Tightly Coupled Memory*.

B3.3.4   TLB Type register

The format of the TLB Type register is:

31 24	23 16	15 8	7 1	0
SBZ/UNP	ILsize	DLsize	SBZ/UNP	S
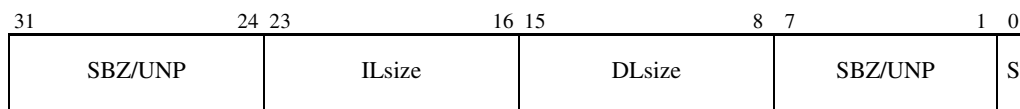
S-bit   Specifies whether the TLB is a unified TLB (S == 0), or separate instruction and data TLBs (S == 1).

DLsize   Specifies the number of lockable entries in the data TLB if S ==1, or the unified TLB if S == 0.

ILsize   Specifies the number of lockable entries in the instruction TLB, if S == 1, otherwise SBZ.

A detailed description of the virtual memory system architecture is provided in Chapter B4 *Virtual Memory System Architecture*.

B3.3.5   MPU Type register

The format of the Memory Protection Unit (MPU) Type register is:

31 24	23 16	15 8	7 1	0
SBZ/UNP	IRegion	DRegion	SBZ/UNP	S
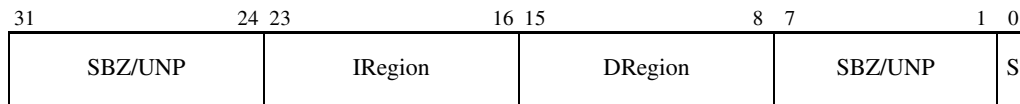
S-bit   Specifies whether the MPU is a unified MPU (S == 0), or separate instruction and data MPUs (S == 1).

DRegion   Specifies the number of protected regions in the data MPU if S ==1, or the unified MPU if S == 0.

IRegion   Specifies the number of protected regions in the instruction MPU, if S == 1, otherwise SBZ.

A detailed description of the protected memory system architecture is provided in Chapter B5 *Protected Memory System Architecture*.

B3.4 Register 1: Control registers

CP15 register 1 contains configuration control bits for the ARM processor. It contains 3 registers selected by the `opcode_2` field. When `opcode_2` is 0 the architecturally specified control register is selected. When `opcode_2` is 1 an IMPLEMENTATION DEFINED control register is selected.

Table B3-3 System Control coprocessor Control registers

opcode2	Register
0b000	Control register
0b001	Auxiliary control register (format IMPLEMENTATION DEFINED)
0b010	Coprocessor access control register
other	RESERVED

B3.4.1 Control register

This register contains:

- Enable/disable bits for the caches, MMUs, and other memory system blocks that are primarily controlled by other CP15 registers. This allows these memory system blocks to be programmed correctly before they are enabled.

- Various configuration bits for memory system blocks and for the ARM processor itself.

——— **Note** ———

Extra bits of both varieties might be added in the future. Because of this, this register should normally be updated using read/modify/write techniques, to ensure that currently unallocated bits are not needlessly modified. Failure to observe this rule might result in code which has unexpected side effects on future processors. One exception that might be useful in some circumstances is that 0 can be written to this register to restore it to its reset state.

31	25 24	16 15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
UNP/SBZP	ARMv6 Update	L4	RR	V	I	Z	F	R	S	B	L	D	P	W	C	A	M
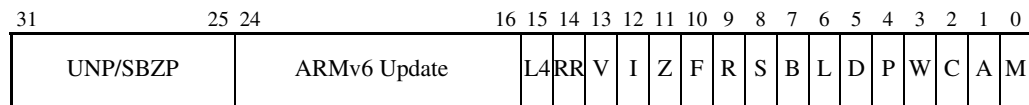
Reading from CP15 register 1 reads the control bits. The `<CRm>` and `<opcode2>` fields are ignored when reading CP15 register 1, and should be zero.

Writing to CP15 register 1 sets the control bits. The `<CRm>` and `<opcode2>` fields are not used when writing CP15 register 1, and should be zero.

When a control bit in CP15 register 1 is not applicable to a particular implementation, it reads as the value that most closely reflects that implementation, and ignores writes. (Specific examples of this general rule are documented in the individual bit descriptions below.) Apart from bits that read as 1 according to this rule, all bits in CP15 register 1 are set to 0 on reset.

M (bit[0])   This is the enable/disable bit for the MMU or Protection Unit:

0 = MMU or Protection Unit disabled

1 = MMU or Protection Unit enabled.

On systems without an MMU, this bit reads as 0 and ignores writes.

A (bit[1])   In ARM architecture v6, this controls strict alignment:

0 = Alignment not strict

1 = Strict alignment. If a data access is not aligned to the width of the accessed data item, a Data Abort exception is generated.

In architectures before v6, for memory systems which optionally allow the alignment of data memory accesses to be checked, this bit enables and disables alignment fault checking:

0 = Alignment fault checking disabled

1 = Alignment fault checking enabled.

For other memory systems, this bit ignores writes, and reads as 1 or 0 according to whether the memory system does or does not check the alignment of data memory accesses.

C (bit[2])   If a L1 unified cache is used, this is the enable/disable bit for the unified cache. If separate L1 caches are used, this is the enable/disable bit for the data cache. In either case:

0 = L1 unified/data cache disabled

1 = L1 unified/data cache enabled.

If the L1 cache is not implemented, this bit reads as 0 and ignores writes. If the L1 cache cannot be disabled, this bit reads as 1 and ignores writes.

The state of this bit does not affect other levels of cache in the system.

W (bit[3])   This is the enable/disable bit for the write buffer:

0 = Write buffer disabled

1 = Write buffer enabled.

If this bit is not implemented, it reads as 1 and ignores writes.

SBO (bits[4:6])

These bits read as 1 and ignore writes.

B (bit[7])   This bit is used to configure the ARM processor to the endianness of the memory system.

ARM processors which support both little-endian and big-endian word-invariant memory systems use this bit to configure the ARM processor to rename the four byte addresses within a 32-bit word.

ARM DDI 0100F-02

In V6 this becomes the mechanism by which legacy big-endian operating systems and applications can be supported.

0 = configured little-endian memory system (LE)

1 = configured big-endian word-invariant memory system (BE-32)

The B bit must be reset by hardware. (Previous architectures allowed an IMPLEMENTATION DEFINED configuration option to pre-set or reset this bit externally, depending on the external memory subsystem).

S (bit[8])   System protection bit, supported for backwards compatibility. The effect of this bit is described in *Access permissions* on page B4-8. The functionality is deprecated in ARMv6.

R (bit[9])   ROM protection bit, supported for backwards compatibility. The effect of this bit is described in *Access permissions* on page B4-8. The functionality is deprecated in ARMv6.

F (bit[10])   The meaning of this bit is IMPLEMENTATION DEFINED.

Z (bit[11])   On ARM processors which support branch prediction, this is the enable/disable bit for branch prediction:

0 = Program flow prediction disabled

1 = Program flow prediction enabled.

If program flow prediction cannot be disabled, this bit reads as 1 and ignores writes. Program flow prediction includes all possible forms of speculative change of instruction stream prediction. Examples include static prediction, dynamic prediction, and return stacks.

On ARM processors that do not support branch prediction, this bit reads as 0 and ignores writes.

I (bit[12])   If separate L1 caches are used, this is the enable/disable bit for the L1 instruction cache:

0 = L1 instruction cache disabled

1 = L1 instruction cache enabled.

If an L1 unified cache is used or the L1 instruction cache is not implemented, this bit reads as 0 and ignores writes. If the L1 instruction cache cannot be disabled, this bit reads as 1 and ignores writes.

The state of this bit does not affect further levels of cache in the system.

V (bit[13])   This bit is used to select the location of the exception vectors:

0 = Normal exception vectors selected (address range 0x00000000-0x0000001C)

1 = High exception vectors selected (address range 0xFFFF0000-0xFFFF001C).

An implementation can provide an input signal that determines the state of this bit after reset.

RR (bit[14])   If the cache allows an alternative replacement strategy to be used that has a more predictable performance, this bit selects it:

0 = Normal replacement strategy (for example, random replacement)

Copyright © 1996–2003 ARM Limited. All rights reserved.
Confidential - Draft

1 = Predictable strategy (for example, round-robin replacement).

L4 (bit[15])   Configure disable load to the PC setting of the T-bit:

0 = LDR PC can set the T-bit

1 = LDR PC does not set the T-bit. This is ARMv4 behavior, for backwards compatibility.

TBD is this still true? If so reword above. the instructions affected by this are:

- *LDM (1)* on page A4-35
- *LDR* on page A4-42
- *POP* on page A7-81

DT (bit[16])   TBD - checking if ARMv6 can declare SBZ for bits[16:20]??

SBO.

SBZ (bit[17])   This bit reads as 0 and ignores writes.

IT (bit[18])   SBO.

SBZ (bit[19])   This bit reads as 0 and ignores writes.

ST (bit[20])   SBZ/UNP.

FI (bit[21])   Configure Fast Interrupt configuration. This bit may be used to reduce interrupt latency in an implementation by disabling IMPLEMENTATION DEFINED performance features:

0 = All performance features enabled

1 = Low interrupt latency configuration enabled.

U(bit[22]))   This bit enables unaligned data access operation, including support for mixed little-endian and big-endian data.

0 = unaligned loads are treated as rotated aligned data accesses (legacy code behavior).

1 = unaligned loads and stores are permitted and mixed-endian data support enabled.

XP(bit[23])   Extended page table configure. This bit configures the hardware page table translation mechanism:

0 = Subpage AP bits enabled.

1 = Subpage AP bits disabled. In this case, hardware translation tables support additional features.

VE(bit[24])   Configure vectored interrupts. Enables use of an IMPLEMENTATION DEFINED hardware mechanism to determine the interrupt vectors:

0 = Interrupt vectors are fixed:

- IRQ at 0x00000018 if V bit == 0, IRQ at 0xFFFF0018 if V bit == 1
- FIQ at 0x0000001C if V bit == 0, FIQ at 0xFFFF001C if V bit == 1

1 = Interrupt vectors are defined by an IMPLEMENTATION DEFINED hardware mechanism.

EE Bit[25]	Mixed Endian exception entry. The EE bit is used to define the value of the CPSR E-bit on entry to an exception vector, including reset. The value is also used to indicate the endianness of page table data for page table lookups. This bit may be preset by CFGEND[1:0] pins on system reset. See *Endian configuration and control* on page A2-33 for more details.
Bits[31:26]	RESERVED. These bits are normally updated using read/modify/write techniques, to ensure that currently unallocated bits are not needlessly modified. Failure to observe this rule might result in code which has unexpected side effects on future processors. One exception that might be useful in some circumstances is that 0 can be written to these bits to restore them to their reset state.

B3.4.2   Auxiliary control register

The format of this register is IMPLEMENTATION DEFINED.

B3.4.3   Coprocessor access register

This register controls accesses to all coprocessors other than CP15 and CP14.

A typical use for this register is to enable an operating system to control coprocessor resource sharing among applications. Initially all applications are denied access to the shared resources. When an application attempts to use that resource it results in an Undefined Instruction exception. The undefined instruction handler can then grant access to that resource by setting the appropriate bit in the coprocessor access register.

Sharing resources among applications requires a state saving mechanism. Two possibilities are:

*   the operating system, during a context switch, saves the state of the coprocessor if the last executing process had access rights to a coprocessor

*   the operating system, after a request for access to a coprocessor, saves off the old coprocessor state with the last process to have access to it.

31	29	27	25	23	21	19	17	15	13	11	9	7	5	3	0
UNP/SBZP	cp13	cp12	cp11	cp10	cp9	cp8	cp7	cp6	cp5	cp4	cp3	cp2	cp1	cp0	

Coprocessor access rights

Each pair of bits corresponds to the access rights for each coprocessor.

00	Access denied. Attempts to access corresponding coprocessor generates an undefined exception.
01	Supervisor access only.
10	RESERVED
11	Full access.

After updating this register an IMB sequence should be executed. None of the instructions executed after changing this register and before the IMB should be coprocessor instructions affected by the change in coprocessor access privileges.

After a system reset all coprocessor access rights are set to *Access denied*.

Any unimplemented coprocessors shall result in the associated bit field read-as-zero (RAZ). This allows system software to write all-1's to the coprocessor access register, then read back the result to determine which coprocessors are present, as part of an auto-configuration sequence

B3.5   Registers 2-15

System Control coprocessor registers other than registers 0 and 1 are allocated to specific areas as follows:

- CP15 registers 2-6, 8, 10, and 13 are allocated to the memory protection system. See Chapter B4 *Virtual Memory System Architecture* and Chapter B8 *Fast Context Switch Extension* for details of these registers.

- CP15 registers 7 and 9 are allocated to the control of caches, and write buffers. See Chapter B6 *Caches and Write Buffers* for details of these registers.

- CP15 register 11 is allocated to the level 1 memory DMA support. See Chapter B7 *Tightly Coupled Memory* for details.

- CP15 register 15 is reserved for IMPLEMENTATION DEFINED purposes. See the technical reference manual for the implementation or other implementation-specific documentation for details of the facilities available through this register.

- CP15 registers 12 and 14 are reserved for future expansion. Accessing (reading or writing) any of these registers is UNPREDICTABLE, and UNDEFINED from ARMv6.

Chapter B4
Virtual Memory System Architecture

This chapter describes the *Virtual Memory System Architecture* (VMSA) based on a *Memory Management Unit* (MMU). It contains the following sections:

B4.1   About the VMSA

Complex operating systems typically use a virtual memory system to provide separate, protected address spaces for different processes. Processes are dynamically allocated memory and other memory mapped system resources under the control of a *Memory Management Unit* (MMU). The MMU allows fine-grained control of a memory system through a set of virtual to physical address mappings and associated memory properties held within one or more structures known as *Translation Lookaside Buffers* (TLBs) within the MMU. The contents of the TLBs are managed through hardware translation lookups from a set of translation tables maintained in memory.

The process of doing a full translation table lookup is called a *translation table walk*. It is performed automatically by hardware, and has a significant cost in execution time, at least one main memory access, and often two. TLBs reduce the average cost of a memory access by caching the results of translation table walks. Implementations can have a unified TLB (von Neumann architecture) or separate Instruction and Data TLBs (Harvard architecture).

The Virtual Memory System Architecture (VMSA) has been significantly enhanced in ARMv6. This is referred to as VMSAv6. To prevent the need for a TLB invalidation on a context switch, each virtual to physical address mapping can be marked as being associated with a particular application space, or as global for all application spaces. Only global mappings and those for the current application space are enabled at any time. By changing the *Application Space IDentifier* (ASID) the enabled set of virtual to physical address mappings can be altered. VMSAv6 has added definitions for different memory types (see section B_TBD_memorder_memtypes), and other attributes (see section B_TBD_mmu_mem_access_crl). For backwards compatibility there is an XP control bit in the System Control Coprocessor, CP15 register1 as defined in section B_TBD_mmu_CP15r1.

The set of memory properties associated with each TLB entry includes:

Memory access permission control

> This controls whether a program has no-access, read-only access, or read/write access to the memory area. When an access is not permitted, a memory abort is signalled to the processor.

> The level of access allowed can be affected by whether the program is running in User mode, or a privileged mode, and by the use of domains.

Memory region attributes

> These describe properties of a memory region. Examples include device (VMSAv6), non-cacheable, write-through, and write-back.

Virtual-to-physical address mapping

> An address generated by the ARM processor is called a *virtual address*. The MMU allows this address to be mapped to a different *physical address*. This physical address identifies which main memory location is being accessed.

> This can be used to manage the allocation of physical memory in many ways. For example, it can be used to allocate memory to different processes with potentially conflicting address maps, or to allow an application with a sparse address map to use a contiguous region of physical memory.

> ――― **Note** ―――
>
> If the *Fast Context Switch Extension* (FCSE) is being used (see TBD xref), all references to virtual addresses in this chapter mean the *modified virtual address* that it generates, except where explicitly stated otherwise. The FCSE is only present in ARMv6 for backwards compatibility. Its use in new systems is deprecated.

System Control coprocessor registers allow high-level control of this system, such as the location of the translation tables. They are also used to provide status information about memory aborts to the ARM.

The VMSA allows for specific TLB entries to be locked down in a TLB. This ensures that accesses to the associated memory areas never require looking up by a translation table walk. This enables the worst case access time to code and data for real-time routines to be minimized and deterministic.

When translation tables in memory are changed or a different translation table is selected (by writing to CP15 register 2), previously cached translation table walk results in the TLBs can cease to be valid. The VMSA therefore supplies operations to flush TLBs.

B4.1.1   Key changes introduced in VMSAv6

The following list summarizes the changes introduced in VMSAv6:

- Entries can be associated with an application space identifier, or marked as a global mapping. This eliminates the requirement for TLB flushes on most context switches.

- Access permissions extended to allow both privileged read only, and privileged/user read-only modes to be simultaneously supported. The use of the System (S) and ROM (R) bits to control access permission determination are only supported for backwards compatibility.

- Memory region attributes to mark pages shared by multiple processors.

- The use of Tiny pages, and the fine page table second level format is now obsolete.

B4.2 Memory access sequence

When the ARM CPU generates a memory access, the MMU performs a lookup for a mapping for the requested virtual address in a TLB. From VMSAv6 this includes the current ASID too. Implementations can use either Harvard or unified TLBs. If the implementation has separate instruction and data TLBs, it uses:

- the instruction TLB for an instruction fetch
- the data TLB for all other accesses.

If no global mapping, or mapping for the currently selected ASID (VMSAv6), for the virtual address can be found in the appropriate TLB then a translation table walk is automatically performed by hardware.

——— **Note** ———

Prior to VMSAv6, all virtual address translations can be considered as globally mapped.

If a matching TLB entry is found then the information it contains is used as follows:

1. The access permission bits and the domain, are used to determine whether access is permitted. If the access is not permitted the MMU signals a memory abort. Otherwise the access is allowed to proceed. TBD xref describes in more detail how this is done.

2. The memory region attributes are used to control:
   - the cache and write buffer
   - whether the access is cached or uncached
   - the target memory type
   - whether the target memory is shared or unshared.

3. The physical address is used for any access to external or tightly coupled memory, and can be used to perform TAG matching for cache entries in physically tagged cache implementations.

Figure B4-1 on page B4-5 shows this for a cached system.
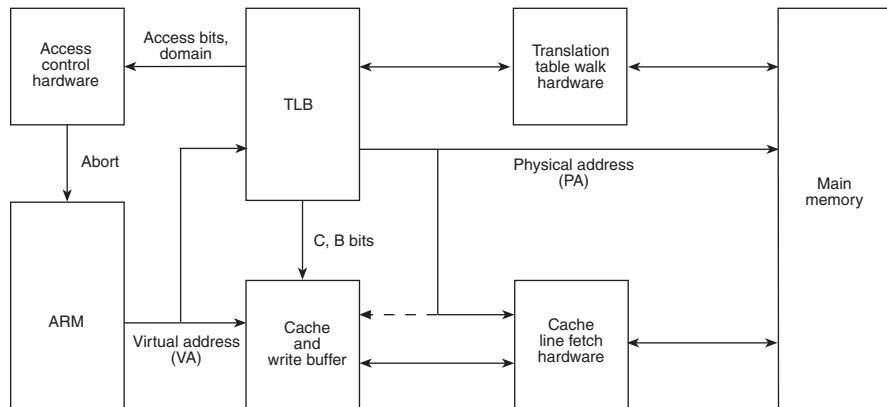
Figure B4-1 Cached MMU memory system overview

B4.2.1   TLB Match Process

Each TLB entry contains a virtual address, a page size, a physical address, and a set of memory properties. It is marked as being associated with a particular application space, or as global for all application spaces. Where an ASID is used, register 13 in CP15 determines the currently selected application space.

A TLB entry matches if bits 31-N of the virtual address match, and it is either marked as global, or the ASID matches the current ASID, where N is $\log_2$ of the page size for the TLB entry.

The behavior of a TLB if two or more entries match at any time (including global and ASID specific entries) is UNPREDICTABLE. The operating system must ensure that at most one TLB entry can match at any time.

A TLB can store entries based on the following block sizes:

Supersections   consist of 16MB blocks of memory

Sections   consist of 1MB blocks of memory

Large pages   consist of 64KB blocks of memory

Small pages   consist of 4KB blocks of memory.

———— **Note** ————

The use of Tiny (1KB) pages is not supported in VMSAv6.

Supersections, sections and large pages are supported to allow mapping of a large region of memory while using only a single entry in a TLB.

If no mapping for an address can be found within the TLB then the translation table is automatically read by hardware, and a mapping is placed in the TLB. See Hardware Pagetable Translation on tbd xref for more details.

B4.2.2 Virtual to physical translation mapping restrictions

The VMSA can be used in conjunction with virtually indexed physically tagged caches. For details of any mapping page table restrictions for virtual to physical addresses see tbd xref.

B4.2.3 Enabling and disabling the MMU

The MMU can be enabled and disabled by writing the M bit (bit[0]) of register 1 of the System Control coprocessor. On reset, this bit is cleared to 0, disabling the MMU.

When the MMU is disabled, memory accesses are treated as follows:

- All data accesses are treated as uncacheable. The value of the C bit (bit[2]) of CP15 register 1 SBZ.

- If a Harvard cache arrangement is used then all instruction accesses are cacheable if the I bit (bit[12]) of CP15 register 1 is set (1), and uncacheable if the I bit is clear (0).

  If a unified cache is used all instruction accesses are treated as uncacheable.

- All explicit accesses are strongly ordered. The value of the W bit (bit[3], write buffer enable) of CP15 register 1 is ignored.

- No memory access permission checks are performed, and no aborts are generated by the MMU.

- The physical address for every access is equal to its virtual address (this is known as a flat address mapping).

- The FCSE PID (see TBD xref) SBZ when the MMU is disabled. This is the reset value for the FCSE PID. If the MMU is to be disabled the FCSE PID should be cleared. The behavior is UNPREDICTABLE if the FCSE is not cleared when the MMU is disabled.

- All change of program flow prediction is disabled. The state of the Z bit (bit [11], branch prediction enable) of CP15 register 1 is ignored. This prevents speculative fetches before the memory region types are defined, protecting read-sensitive I/O locations.

- All MMU and Cache CP15 operations work as normal when the MMU is disabled.

- Instruction and data prefetch operations work as normal.

  ——— **Note** ———

  The D-cache cannot be enabled when the MMU is disabled, and therefore a data prefetch operation has no effect. Instruction prefetch operations have no effect if the instruction cache is disabled. If the instruction cache is enabled they perform a prefetch to the instruction cache. No memory access permissions are performed, and the address is flat mapped.

  ————————

- Accesses to the TCMs work as normal if the TCM is enabled.

Before the MMU is enabled all relevant CP15 registers must be programmed. This includes setting up suitable translation tables in memory. Prior to enabling the MMU the instruction cache should be disabled and invalidated. The instruction cache can then be re-enabled at the same time as the MMU is enabled.

——— **Note** ———

Enabling or disabling the MMU effectively changes the virtual-to-physical address mapping (unless the translation tables are set up to implement a flat address mapping). Any virtually tagged caches, for example, that are enabled at the time need to be flushed (see *Address mapping changes* on page B2-22).

In addition, if the physical address of the code that enables or disables the MMU differs from its virtual address, instruction prefetching can cause complications (see *Prefetching and self-modifying code* on page B2-24). It is therefore strongly recommended that code which enables or disables the MMU has identical virtual and physical addresses.

B4.3   Memory access control

Access to a memory region is controlled by the access permission and domain bits in the TLB entry. APX and XN (execute never) bits have been added in VMSAv6. These form part of the page table entry formats described in section B_TBD_mmu_pagetabtranslation.

B4.3.1   Access permissions

The access permission bits control access to the corresponding memory region. If an access is made to an area of memory without the required permissions, then a Permission Fault is raised. The access permissions are determined by a combination of the AP and APX bits in the page table, and the S and R bits in CP15 register 1. For page table formats not supporting the APX bit, the value 0 is used.

—— **Note** ——

The use of the S and R bits is deprecated in VMSAv6. Changes to the S and R bits do not affect the access permissions of entries already in the TLB. The TLB must be flushed for the updated S and R bit values to take effect.

If an access is made to an area of memory without the required permission, a Permission Fault is raised (see *Aborts* on page B4-14).

Table Table B4-1 shows the encoding of the access permissions.

Table B4-1 MMU access permissions

S	R	APX ^a	AP[1:0]	Privileged permissions	User permissions	Description
0	0	0	0b00	No access	No access	All accesses generate permission faults
x	x	0	0b01	Read/write	No access	Privileged access only
x	x	0	0b10	Read/write	Read only	Writes in User mode generate permission faults
x	x	0	0b11	Read/write	Read/write	Full access
0	0	1	0b00	-	-	RESERVED
0	0	1	0b01	Read only	No access	Privileged read only
0	0	1	0b10	Read only	Read only	Privileged/User read only
0	0	1	0b11	-	-	RESERVED
0	1	0	0b00	Read only	Read only	Privileged/User read only
1	0	0	0b00	Read only	No access	Privileged read only
1	1	0	0b00	-	-	RESERVED
0	1	1	0bxx	-	-	RESERVED
1	0	1	0bxx	-	-	RESERVED
1	1	1	0bxx	-	-	RESERVED

a. VMSAv6 and above only.

Each memory region can be tagged as not containing executable code. If the *Execute-Never* (XN) bit is set to 1, any attempt to execute an instruction in that region results in a permission fault. If the XN bit is cleared to 0, code can execute from that memory region.

——— **Note** ———

The XN bit acts as an additional permission check. The address must also have a valid read access.

B4.3.2   Domains

A domain is a collection of memory regions. The ARM architecture supports 16 domains. Each page table entry and TLB entry contains a field that specifies which domain the entry is in. Access to each domain is controlled by a two-bit field in the Domain Access Control Register. Each field allows the access to an entire domain to be enabled and disabled very quickly, so that whole memory areas can be swapped in and out of virtual memory very efficiently. Two kinds of domain access are supported:

Clients   Are users of domains (execute programs and access data), and are guarded by the access permissions of the TLB entries for that domain.

Managers   Control the behavior of the domain (the current sections and pages in the domain, and the domain access), and are not guarded by the access permissions for TLB entries in that domain.

One program can be a client of some domains, and a manager of some other domains, and have no access to the remaining domains. This allows very flexible memory protection for programs that access different memory resources. Table B4-2 shows the encoding of the bits in the Domain Access Control Register.

Table B4-2 Domain Access Values

Value	Access types	Description
0b00	No access	Any access generates a domain fault
0b01	Client	Accesses are checked against the access permission bits in the TLB entry
0b10	Reserved	Using this value has UNPREDICTABLE results
0b11	Manager	Accesses are not checked against the access permission bits in the TLB entry, so a permission fault cannot be generated

B4.4   Memory region attributes

Each TLB entry has an associated set of memory region attributes. These control accesses to the caches, how the write buffer is used, and if the memory region is shareable and therefore must be kept coherent.

Prior to VMSAv6, only C (cacheable) and B (bufferable) bits were provided. Their exact usage model (for example, how the bit settings affected write through versus write back cache policies) and any additional control were IMPLEMENTATION DEFINED. VMSAv6 has introduced a more formal memory model (see chapter B_TBD_memorder), supported by the additional bit field (TEX) and definitions described in this section.

B4.4.1   C, B, and TEX Encodings

Page table formats use five bits to encode the memory region type. These are TEX[2:0] and the C and B bits. Table B4-3 on page B4-12 shows the mapping of the *Type extension* field (TEX) and the cacheable and bufferable bits (C and B) to memory region type. For page tables formats with no TEX field the value 0b000 is used.

In addition, certain page tables contain the shared bit (S). This bit only applies to normal, not device or strongly ordered memory, and determines if the memory region is shared (1), or not-shared (0). If not present the S bit is assumed to be 0 (not-shared).

Table B4-3 shows the C, B, and TEX encodings.

Table B4-3 CB + TEX Encodings

TEX	C	B	Description	Memory type	Page shareable
0b000	0	0	Strongly ordered	Strongly ordered	Shareable
0b000	0	1	Shared Device	Device	Shareable
0b000	1	0	Outer and inner write through, no write allocate	Normal	S
0b000	1	1	Outer and inner write back, no write allocate	Normal	S
0b001	0	0	Outer and inner non-cacheable	Normal	S
0b001	0	1	RESERVED	-	-
0b001	1	0	IMPLEMENTATION DEFINED	IMPLEMENTATION DEFINED	IMPLEMENTATION DEFINED
0b001	1	1	Outer and inner write back, write allocate	Normal	S
0b010	0	0	Non-shared device	Device	Not shareable
0b010	0	1	RESERVED	-	-
0b010	1	X	RESERVED	-	-
0b011	X	X	RESERVED	-	-
0b100	X	X	RESERVED	-	-
0b1BB	A	A	Cached memory BB = outer policy, AA = inner policy	Normal	S

S indicates shareable if page table present, and S-bit in page table set, otherwise not shareable.

For an explanation of the Sharable attribute, and Normal, Strongly ordered and Device memory types see tbd xref

The terms Inner and Outer refer to levels of caches that might be built in a system. Inner refers to the innermost caches, including Level 1. Outer refers to the outermost caches. The boundary between Inner and Outer Caches is defined in the implementation of a cached system. Inner always includes L1. For example, in a system with three levels of caches, the Inner attributes might apply to L1 and L2, whereas the Outer attributes apply to L3. In a two-level system, it is expected that Inner applies to L1 and Outer to L2.

Table B4-4 shows the encoding of the inner and outer cache policies.

Table B4-4 Inner and outer cache policy

Encoding		Description
0	0	Non-cacheable
0	1	Write back, write allocate
1	0	Write through, no write allocate
1	1	Write back, no write allocate

It is optional which write allocation policies an implementation supports. The *allocate on write* and *no allocate on write* cache policies indicate which allocation policy is preferred for a memory region, but it should not be relied on that the memory system implements that policy.

Not all inner and outer cache policies are mandatory. Table B4-5 describes the implementation options.

Table B4-5 Cache policy implementation options

Cache policy	Implementation options
Inner non-cacheable	Mandatory.
Inner write through	Mandatory.
Inner write back	Optional. If not supported, memory system should implement as inner write through.
Outer non-cacheable	Mandatory.
Outer write through	Optional. If not supported, memory system should implement as outer non-cacheable.
Outer write back	Optional. If not supported, memory system should implement as outer write through.

B4.5 Aborts

Mechanisms that can cause the ARM processor to take an exception because of a memory access are:

MMU fault   The MMU detects the restriction and signals the processor.

Debug abort   Monitor debug-mode is enabled and a breakpoint or a watchpoint has been detected.

External abort   The external memory system signals an illegal or faulting memory access.

Collectively, these are called *aborts*. Accesses that cause aborts are said to be aborted, and use Fault Address and Fault Status registers to record associated context information. the FAR and FSR registers are described in *Fault address and fault status registers* on page B4-18

B4.5.1 MMU faults

The MMU generates four types of fault:
- alignment fault
- translation fault
- domain fault
- permission fault.

Aborts that are detected by the MMU are stopped before any external memory access takes place.

If the memory request that aborts is an instruction fetch, then a Prefetch Abort exception is raised if and when the processor attempts to execute the instruction corresponding to the aborted access. If the aborted access is a data access or a cache maintenance operation, a Data Abort exception is raised. See *Exceptions* on page A2-15 for more information about Prefetch and Data Aborts.

Fault-checking sequence

The sequence used by the MMU to check for access faults is slightly different for Sections and Pages. Figure B4-2 on page B4-15 shows the sequence for both types of access.

Figure B4-2 Sequence for checking faults

Alignment fault

For details of when alignment faults are generated, see section A_TBD_ch2_alignment

Translation fault

There are two types of translation fault:

Section   This is generated if the first-level descriptor is marked as invalid. This happens if bits[1:0] of the descriptor are both 0, and in VMSAv6 formats when the value is 0b11, a RESERVED value.

Page   This is generated if the second-level descriptor is marked as invalid. This happens if bits[1:0] of the descriptor are both 0.

Domain fault

There are two types of domain fault:

Section domain faults
    the domain is checked when the first-level descriptor is returned

Page domain faults
    the domain is checked when the second-level descriptor is returned.

Permission fault

If the two-bit domain field returns *client* (01), the permission access check is performed on the access permission field in the TLB entry.

B4.5.2   Debug events

When Monitor debug-mode is enabled an abort can be taken because of a breakpoint on an instruction access or a watchpoint on a data access. In both cases the memory system completes the access before the abort is taken.

If an abort is taken because of Monitor debug-mode then the appropriate FSR (instruction or data) is updated to indicate a Debug abort. This is the only information saved on a prefetch abort (a breakpoint) debug event. This is a precise abort. R14_abt is used to determine the address of the failing instruction.

Watchpoints are not taken precisely, because following instructions can run underneath load and store multiples. The debugger must read the WFAR to determine which instruction caused the debug event.

B4.5.3   External aborts

External memory errors are defined as those that occur in the memory system other than those that are detected by an MMU. External memory errors are expected to be rare and are likely to be fatal to the running process. An example of an event that could cause an external memory error is an uncorrectable parity or ECC failure on a Level 2 Memory structure.

It is IMPLEMENTATION DEFINED which, if any, external aborts are supported.

The presence of a precise external abort is signalled in the DFSR or IFSR. For further details of the imprecise external abort model see *Imprecise aborts* on page A2-22.

External abort on instruction fetch

Externally generated errors during an instruction prefetch are precise in nature, and are only recognized by the CPU if it attempts to execute the instruction fetched from the location that caused the error. The resulting failure is reported in the IFSR if no higher priority abort (including a Data Abort) has taken place.

The Fault Address Register is not updated on an external abort on instruction fetch.

External abort on data read/write

Externally generated errors during a data read or write can be imprecise. This means that R14_ABORT on entry into the Abort handler on such an abort is not guaranteed to hold an address that is related to the instruction that caused the exception. Correspondingly, external aborts can be unrecoverable.

If an imprecise external abort causes entry into the abort state while the abort state is not re-entrant, then the processor is in an unrecoverable state as the R14 and SPSR values has been corrupted. For this reason, the existence of an imprecise external abort must only be recognized by the processor at a point when the abort state is re-entrant. This is managed by the provision of a mask for imprecise external aborts in the CSPR, which is referred to as the A bit.

This mask, and the associated requirements for holding pending imprecise external aborts are described in tbd xref. Entry into the abort state caused by an imprecise external abort causes the data FSR to indicate the presence of an imprecise external abort.

The FSR is not updated on an imprecise external abort on a data access.

External abort on a hardware page table walk

An external abort occurring on a hardware page table access must be returned with the page table data. Such aborts are precise. The FAR is updated on an external abort on a hardware page table walk on a data access, but not on an instruction access. The appropriate FSR indicates that this has occurred.

B4.6 Fault address and fault status registers

Prior to VMSAv6, the architecture supported a single *Fault Address Register* (FAR) and *Fault Status Register* (FSR).

VMSAv6 requires four registers:

- an *Instruction Fault Status Register* (IFSR) updated on prefetch aborts

- a *Data Fault Status Register* (DFSR) updated on data aborts

- a *Fault Address Register* (FAR) updated with the faulting address for precise exceptions

- a *Watchpoint Fault Address Register* (WFAR) updated on a watchpoint access with the address of the instruction that caused the data abort.

—— **Note** ——

The IFSR and DFSR are updated on data aborts because of instruction cache maintenance operations.

For a description of precise and imprecise exceptions see A_TBD_ch2_exceptions.

VMSAv6 added a fifth fault status bit (bit[10]) to both the IFSR and DFSR. It is IMPLEMENTATION DEFINED how this bit is encoded in earlier versions of the architecture. A write flag (bit[11] of the DFSR) has also been introduced.

Precise aborts resulting from data accesses (Precise Data Aborts) are immediately acted upon by the CPU. The DFSR is updated with a four-bit Fault Status (FS[3:0]) and the domain number of the access. In addition, the virtual address which caused the Data Abort is written into the FAR. If a data access simultaneously generates more than one type of Data Abort, they are prioritized in the order given in Table B4-6 on page B4-19.

Aborts arising from instruction fetches are flagged as the instruction enters the instruction pipeline. Only when, and if, the instruction is executed does it cause a Prefetch Abort exception. An abort resulting from an instruction fetch is not acted upon if the instruction is not used (for example, if it is branched around). Normally, the fault address associated with a Prefetch Abort exception is determined from the value saved in R14_abt when the Prefetch Abort exception vector is entered.

It is IMPLEMENTATION DEFINED whether the FSR and FAR are updated for an abort arising from an instruction fetch, and if so, what useful information they contain about the fault. However, an abort arising from an instruction fetch never updates the FSR and the FAR between the time that an abort arising from a data access updates them and the time of the corresponding entry into the Data Abort exception vector. In other words, a Data Abort handler can rely upon its FAR and FSR values not being corrupted by an abort arising from an instruction fetch that was not acted upon. From VMSAv6, only the IFSR is updated by a prefetch abort.

Table B4-6 Fault status register encodings

Architecture	Priority	Sources		FS [3:0]	Domain	FAR
All	Highest	Alignment		0b00001	Invalid	Valid
VMSAv6		PMSA - TLB miss (MPU)		0b00000	Invalid	Valid
		Alignment (deprecated)		0b00011		
VMSAv6		Instruction Cache Maintenance Operation Fault		0b00100	Invalid	Valid
All		External Abort on Translation	1st level	0b01100	Invalid	Valid
			2nd level	0b01110	Valid	Valid
All		Translation	Section	0b00101	Invalid	Valid
			Page	0b00111	Valid	Valid
All		Domain	Section	0b01001	Valid	Valid
			Page	0b01011	Valid	Valid
All		Permission	Section	0b01101	Valid	Valid
			Page	0b01111	Valid	Valid
VMSAv6		Precise External Abort		0b01000	Invalid	Valid
		External Abort, Page (deprecated)		0b01010		
VMSAv6		Imprecise External Abort		0b10110	Invalid	Invalid
VMSAv6		Parity Error Exception		0b01000	Invalid	Invalid
VMSAv6	Lowest	Debug event		0b00010	Valid	UNPREDICT-ABLE

B4.6.1 Notes for fault status register encodings table

Prior to VMSAv6, the usage of FS[3:0] values associated with items marked as ARMv6 is IMPLEMENTATION DEFINED. This is true for either value of FS[10].

All other FS encodings are RESERVED.

Domain and Fault Address information is only available for data accesses. For instruction aborts R14 must be used to determine the faulting address. The domain information can be determined by performing a TLB lookup for the faulting address and extracting the domain field.

From VMSAv6:

- All data aborts cause the Data Fault Status Register (DFSR) to be updated so that the cause of the abort can be determined. All instruction aborts cause the Instruction Fault Status Register (IFSR) to be updated so that the cause of the abort can be determined.

- For all data aborts, excluding external aborts (other than on translation), the Fault Address Register (FAR) will be updated with the address that caused the abort. External data aborts, other than on translation, can all be imprecise and hence the FAR does not contain the address of the abort. See section *Imprecise aborts* on page A2-22 for more details on imprecise aborts.

- If a translation abort occurs during a data cache maintenance operation by virtual address, a Data Abort is taken and the DFSR indicates the reason. The FAR provides the faulting address.

- If a translation abort occurs during an instruction cache maintenance operation by virtual address, then a Data Abort is taken, and a Instruction Cache Maintenance Operation Fault indicated in the DFSR. The IFSR indicates the reason. The FAR provides the faulting address.

- Where the FCSE mechanism (see Chapter B8 *Fast Context Switch Extension*) is enabled, the WFAR contains the virtual address of the instruction causing the abort, not the modified virtual address.

- The WFAR is used to store the address of the instruction that caused the watchpoint access.

B4.6.2   Abort FSR/FAR update summary

For VMSAv6, a summary of which abort vector is taken, and which of the fault status and fault address registers are updated on each abort type is given in Table B4-7.

Table B4-7 Abort FSR/FAR update summary

Abort Type	Vector	Precise	IFSR	DFSR	FAR	WFAR
Instruction MMU fault	PABORT	Y	X	-	-	-
Instruction debug abort	PABORT	Y	X	-	-	-
Instruction external abort on translation	PABORT	Y	X	-	-	-
Instruction external abort	PABORT	Y	X	-	-	-
Instruction cache Parity error	PABORT	Y	X	-	-	-
Instruction cache maintenance operation	DABORT	Y	X	X	X	-
Data MMU fault	DABORT	Y	-	X	X	-
Data debug abort	DABORT	N	-	X	-	X
Data external abort on translation	DABORT	Y	-	X	X	-
Data external abort	DABORT	N	-	X	-	-
Data cache Parity error	DABORT	N	-	X	-	-
Data cache maintenance operation	DABORT	Y	-	X	X	-

X   Register is updated on this abort type

-   Register is not updated on this abort type.

Copyright © 1996–2003 ARM Limited. All rights reserved.
Confidential - Draft

B4.7 Hardware pagetable translation

The MMU supports memory accesses based on sections or pages:

Supersections (optional)

Consist of 16MB blocks of memory

Sections   Consist of 1MB blocks of memory.

The following page sizes are supported:

Tiny pages (not in VMSAv6)

Consist of 1KB blocks of memory.

Small pages   Consist of 4KB blocks of memory.

Large pages   Consist of 64KB blocks of memory.

Sections and large pages are supported to allow mapping of a large region of memory while using only a single entry in the TLB. Additional access control mechanisms are extended within small pages to 1KB subpages, and within large pages to 16KB subpages. The use of subpage AP bits is deprecated in VMSAv6 and only available in the backwards compatible table entry format, see section B_TBD_mmu_vmsav6_formats for more details. Access control for tiny pages apply to the whole page.

The translation table held in main memory has two levels:

First-level table   Holds section and supersection translations, and pointers to second-level tables.

Second-level tables   Hold both large and small page translations. A second form of page table, fine rather than coarse, supports tiny pages.

The MMU translates virtual addresses generated by the CPU into physical addresses to access external memory, and also derives and checks the access permission. Translations occur as the result of a TLB miss, and start with a first-level fetch. A section-mapped access only requires a first-level fetch, whereas a page-mapped access also requires a second-level fetch.

B4.7.1 Translation table base

The translation process is initiated when the on-chip TLB does not contain an entry for the requested virtual address. The *Translation Table Base Register* (TTBR in CP15 register 2) holds the physical address of the base of the first-level table.

Prior to VMSAv6, a single TTBR existed. Only bits[31:14] of the Translation Table Base Register are significant, and bits[13:0] should be zero. Therefore, the first-level page table must reside on a 16KB boundary.

VMSAv6 introduced an additional translation table base register and a translation table base control register: TTBR0, TTBR1 and TTBCR. On a TLB miss, the top bits of the virtual address determine if the first or second translation table base is used, see section B_TBD_mmu_TTBx for a detailed description including TTBR1 address restrictions.

TTBR1 is expected to be used for operating system and I/O addresses, which do not change on a context switch. TTBR0 is expected to be used for process specific addresses. When TTBCR is programmed to zero, all translations use TTBR0 in a manner compatible with earlier versions of the architecture. The size of the TTBR0 table is always 16KB, but the TTBR1 table ranges in size from 128 bytes to 16KB, depending on the value in the TTBCR. All translation tables must be naturally aligned.

VMSAv6 has also introduced a control bit field into the lowest bits of the TTBRs.

B4.7.2 First-level fetch

Bits[31:14] of the Translation Table Base register are concatenated with bits[31:20] of the virtual address and two zero bits to produce a 32-bit physical address as shown in Figure B4-3. This address selects a four-byte translation table entry which is a first-level descriptor for a section or a pointer to a second-level page table.
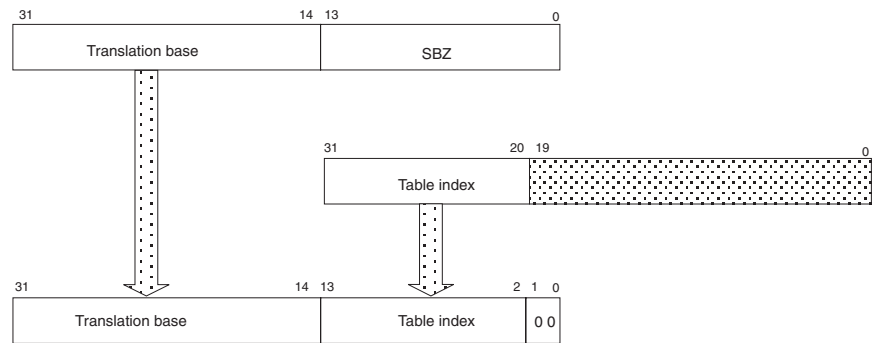
Figure B4-3 Accessing the translation table first-level descriptors

— **Note** —

In some VMSAv6 systems, two translation table base registers are used. In this case, bits [31:14-N] of TTBR0 are concatenated with bits [31-N:20] of the virtual address, where 0<=N<=7, as described in section B_TBD_mmu_TTB

B4.7.3 Pagetable translation in VMSAv6

VMSAv6 supports two page table formats:

- A backwards compatible format supporting sub-page access permissions. These have been extended so certain page table entries support extended region types.

- A new format, not supporting sub-page access permissions, but with support for ARMv6 MMU features. These features are:
    — extended region types
    — global and process specific pages

Copyright © 1996–2003 ARM Limited. All rights reserved.
Confidential - Draft

— more access permissions
— marking of shared and not-shared regions
— marking of execute-never regions.

Subpages are described in section B_TBD_mmu_L2desc.

It is IMPLEMENTATION DEFINED whether hardware page table walks can cause a read from the L1 unified/data cache. Hardware page table walks cannot cause reads from TCM. The RGN, P, S and C bits in the translation table base registers determine the memory region attributes for the page table walk. To ensure coherency on implementations that do not support page tables accesses from L1, either page tables should be stored in inner write-through memory, or if in inner write-back, the appropriate cache entries must be cleaned after modification. Page table walks may be outer-cacheable accessible as defined in the TTBR region (RGN) bits on page TBD.

The page table format is selected using the XP bit in CP15 register 1. When subpage AP bits are enabled (CP15 register1 XP = 0) the page table formats are backwards compatible with ARMv4/v5:

• all mappings are treated as global, and executable (XN = 0)
• all normal memory is non-shared
• device memory may be shared or not-shared as determined by the TEX + CB bits
• the use of subpage AP bits where AP3, AP2, AP1, AP0 contain different values is deprecated.

When subpage AP bits are disabled (CP15 register1 XP = 1), the page tables have support for ARMv6 MMU features. New page table bits are added to support these features:

• The not-global (nG) bit, determines if the translation should be marked as global (0), or process specific (1) in the TLB. For process specific translations the translation is inserted into the TLB using the current ASID, from the ContextID register.

• The shared (S) bit, determines if the translation is for not-shared (0), or shared (1) memory. This only applies to normal memory regions. Device memory can be shared or not-shared as determined by the TEX + CB bits.

• The execute-never (XN) bit determines whether the region is executable (0) or not-executable (1).

• Three access permission bits. The access permissions extension (APX) bit provides an extra access permission bit.

• All page table mappings support the TEX field.

——— **Note** ———

In VMSAv6, an invalid entry (bits[1:0] = 0b00) or a RESERVED entry (bits[1:0] = 0b11) shall result in a translation fault.

The following sections describe the first and second level access mechanisms, and define the different table formats for VMSAv6 and earlier versions of the architecture.

B4.7.4 First-level descriptors

Each entry in the first-level table is a descriptor of how its associated 1MB virtual address range is mapped. Bits[1:0] of the first-level page table entry determine the type of first-level descriptor as follows:

- If bits[1:0] == 0b00, the associated virtual addresses are unmapped, and attempts to access them generate a translation fault (see *Aborts* on page B4-14). Software can use bits[31:2] for its own purposes in such a descriptor, as they are ignored by the hardware. Where appropriate, it is suggested that bits[31:2] continue to hold valid access permissions for the descriptor.

- If bits[1:0] == 0b10, the entry is a section descriptor for its associated virtual addresses. See *Sections and supersections* on page B4-27 for details of how it is interpreted.

- If bits[1:0] == 0b01, the entry gives the physical address of a coarse second-level table, that specifies how the associated 1MB virtual address range is mapped. Coarse tables require 1KB per table. and can only map large pages and small pages, whereas fine tables can map large and small pages, see *Coarse page table descriptor* on page B4-29.

- If bits[1:0] == 0b11, the entry gives the physical address of a fine second-level table prior to VMSAv6, and is RESERVED in VMSAv6. A fine second-level page table specifies how the associated 1MB virtual address range is mapped. It requires 4KB per table, and can map large, small and tiny pages, see *Fine page table descriptor (ARMv4/v5 only)* on page B4-30.

There are three formats of first-level descriptor table:

- the format used prior to VMSAv6 shown in Table B4-8
- VMSAv6 backwards compatible format shown in Table B4-9 on page B4-26
- VMSAv6 revised format shown in Table B4-10 on page B4-26.

The AP, APX and domain fields are described in section B_TBD_mmu_access. The C, B and TEX fields are described in section B_TBD_mmu_memregion.

The P bit indicates to the system memory controller that this region must have ECC enabled, if supported. It is IMPLEMENTATION DEFINED whether this functionality is implemented prior to VMSAv6.

Table B4-8 First-level descriptor format (prior to VMSAv6)

	31 20	19 14	12 11	10 9	8 5	4 3 2	1 0
Fault	IGN						0 0
Coarse page table	Coarse page table base address			S B Z	Domain	IMP	0 1
Section	Section base address	SBZ	AP	S B Z	Domain	I M C B P	1 0
Fine page table	Fine page table base address		SBZ	Domain	IMP		1 1

       Copyright © 1996–2003 ARM Limited. All rights reserved.
Confidential - Draft

Table B4-9 First-level descriptor format (VMSAv6, backwards compatible)

	31 20	19 14	12 11 10	9	8 5	4 3 2	1 0
Fault	IGN						0 0
Coarse page table	Coarse page table base address			P	Domain	SBZ	0 1
Section	Section base address	SBZ	TEX AP	P	Domain	SBZ C B	1 0
	RESERVED						1 1

Table B4-10 First-level descriptor format (VMSAv6, revised)

	31 24	23 20	19	14 12 11 10	9	8 5	4 3 2	1 0
Fault	SBZ							0 0
Coarse page table	Coarse page table base address				P	Domain	SBZ	0 1
Section	Section base address	SBZ 0 nG S APX		TEX AP	P	Domain	XN C B	1 0
Supersection	Supersection base address	Base address [35:32]	SBZ 1 nG S APX	TEX AP	P	Base Domain address [39:36]	XN C B	1 0
	RESERVED							1 1

B4.7.5   Sections and supersections

If Bits[1:0] equal 0b10, the first-level descriptor is a 1MB section or a 16MB supersection descriptor.

Supersections are optional. If used, they translate 32-bit virtual addresses to a larger physical address space (up to 8 additional address bits), and are defined as follows:

*   The bit fields are described in the VMSAv6 revised format, see *First-level descriptor format (VMSAv6, revised)* on page B4-26.

Bits[1:0]	= 0b10
Bit[18]	= 0 defines a 1MB section
	= 1 defines a 16MB supersection
Bits[8:5]	optional extended physical address bits; PA[39:36]
Bits[23:20]	optional extended physical address bits; PA[35:32].

*   It is IMPLEMENTATION DEFINED how many additional address bits are defined.

*   Supersections default to domain 0.

*   It is IMPLEMENTATION DEFINED whether supersections are offered in section descriptor formats prior to ARMv6.

Figure B4-4 on page B4-28 shows how virtual to physical addresses are generated for sections. The shaded area of the descriptor represents the access control data fields.

——— **Note** ———

The access permissions in the first-level descriptor must be checked before the physical address is generated. The sequence for checking access permissions is given in *Access permissions* on page B4-8.
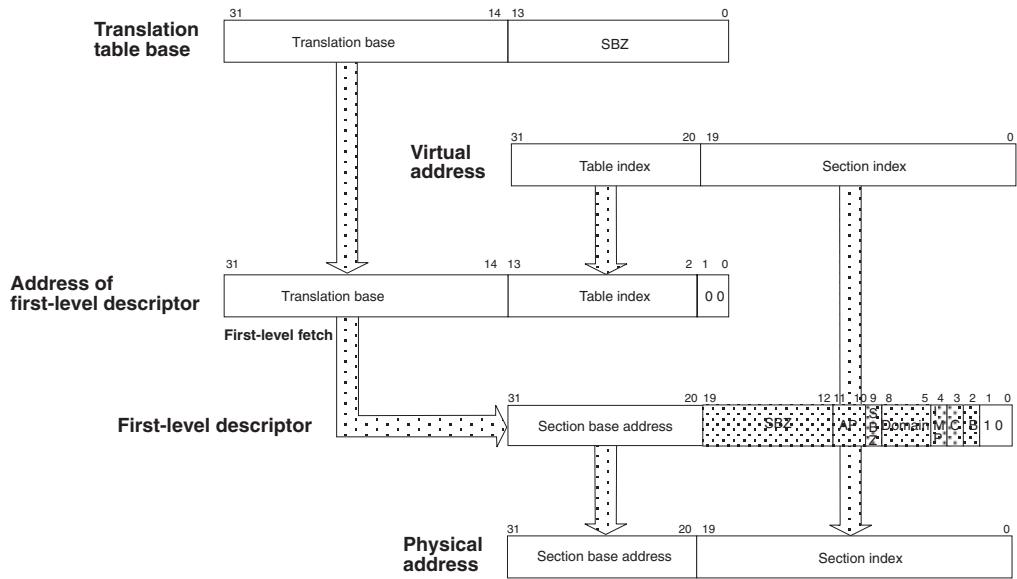
Figure B4-4 Section translation

ARM DDI 0100F-02

B4.7.6   Coarse page table descriptor

If the first-level descriptor is a coarse page table descriptor, the fields have the following meanings:

Bits[1:0]   Identify the type of descriptor (0b01 marks a coarse page table descriptor).

Bits[4:2]   The meaning of these bits is IMPLEMENTATION DEFINED. From VMSAv6 these bits SBZ.

Bits[8:5]   The domain field specifies one of the sixteen possible domains for all the pages controlled by this descriptor.

Bit[9]   This bit can be defined as the P bit. This indicates that ECC should be enabled when 1, otherwise SBZ.

Bits[31:10]   The Page Table Base Address is a pointer to a coarse second-level page table, giving the base address for a second-level fetch to be performed. Coarse second-level page tables must be aligned on a 1KB boundary.

If a coarse page table descriptor is returned from the first-level fetch, a second-level fetch is initiated to retrieve a second-level descriptor, as shown in Figure B4-5. The shaded area of the descriptor represents the access control data fields.
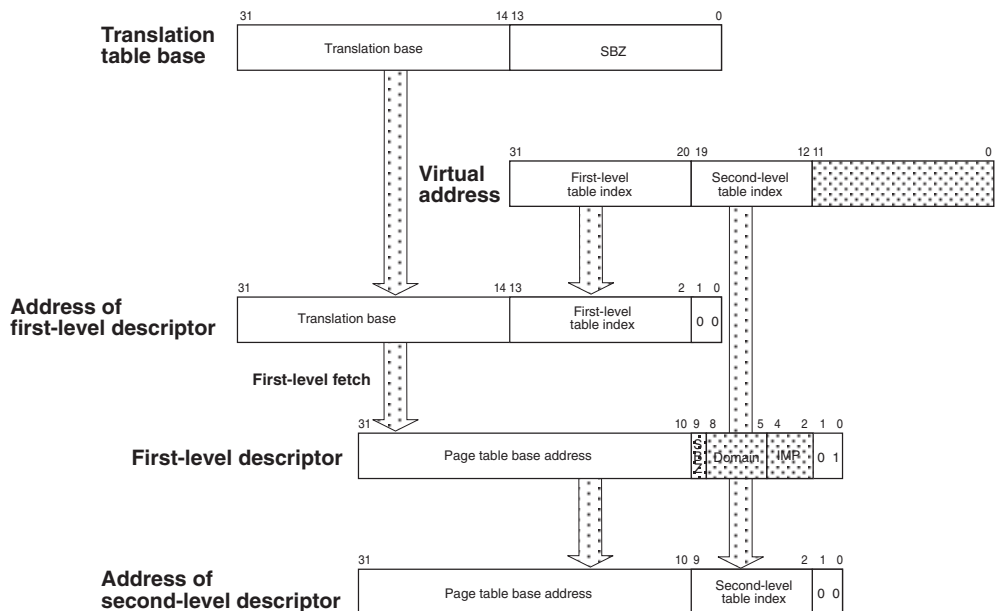
Figure B4-5 Accessing coarse page table second-level descriptors

B4.7.7   Fine page table descriptor (ARMv4/v5 only)

If the first-level descriptor is a fine page table descriptor, the fields have the following meanings:

Bits[1:0]   Identify the type of descriptor (0b11 marks a fine page table descriptor).

Bits[4:2]   The meaning of these bits is IMPLEMENTATION DEFINED.

Bits[8:5]   The domain field specifies one of the sixteen possible domains for all the pages controlled by this descriptor.

Bit[11:9]   These bits are not currently used, and should be zero.

Bits[31:12]   The Page Table Base Address is a pointer to a fine second-level page table, giving the base address for a second-level fetch to be performed. Fine second-level page tables must be aligned on a 4KB boundary.

If a fine page table descriptor is returned from the first-level fetch, a second-level fetch is initiated to retrieve a second-level descriptor, as shown in Figure B4-6. The shaded area of the descriptor represents the access control data fields.
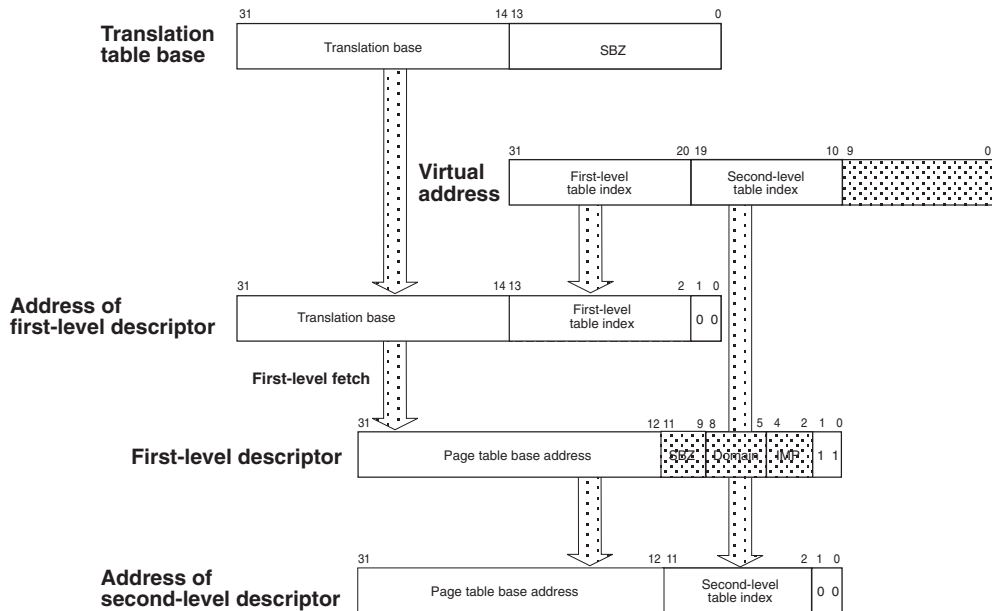
Figure B4-6 Accessing fine page table second-level descriptors

ARM DDI 0100F-02

B4.7.8   Second-level descriptor

There are two forms of second-level page table:

- Coarse tables, 1KB in size. Each 32-bit entry provides translation information for 4KB of memory.
- Fine tables, 4KB in size. Each 32-bit entry provides translation information for 1KB of memory.

The VMSA supports three page sizes:

- large pages are 64KB in size
- small pages are 4KB in size
- tiny pages are 1KB in size.

A second-level table can support page sizes greater than or equal to the amount of memory mapped by an entry. For this reason, tiny pages are only supported in fine page tables. To map pages larger than the entry size, the page table entry needs to be replicated in the table the appropriate number of times:

- for a fine table, small pages require four replicated entries and large pages 64 replicated entries
- for a coarse table, large pages require 16 replicated entries.

---- **Note** ----

Fine second-level page tables are only supported in ARMv4 and ARMv5. Fine page tables and tiny pages are OBSOLETE in VMSAv6.

There are three formats of second-level descriptor table:

- the format used prior to VMSAv6 shown in Table B4-11
- VMSAv6, backwards compatible format shown in Table B4-12 on page B4-32
- VMSAv6, revised format shown in Table B4-13 on page B4-32.

Table B4-11 Second-level descriptor format (prior to VMSAv6)

	31 16	15 12	11 10	9 8	7 6	5 4	3	2	1	0
Fault	IGN								0	0
Large page	Large table base address	SBZ	AP3	AP2	AP1	AP0	C	B	0	1
Small page	Small page base address		AP3	AP2	AP1	AP0	C	B	1	0
Tiny page	Tiny page base address		SBZ		AP	C	B	1	1

Table B4-12 Second-level descriptor format (VMSAv6, backwards compatible)

	31 … 16	15	14 … 12	11 10	9 8	7 6	5 4	3	2	1	0
Fault	IGN									0	0
Large page	Large table base address	SBZ	TEX	AP3	AP2	AP1	AP0	C	B	0	1
Small page	Small page base address			AP3	AP2	AP1	AP0	C	B	1	0
Extended small page	Extended small page base address			SBZ	TEX		AP	C	B	1	1

Table B4-13 Second-level descriptor format (VMSAv6, revised)

	31 … 16	15	14 … 12	11	10	9	8 … 6	5 4	3	2	1	0
Fault	IGN										0	0
Large page	Large table base address	XN	TEX	nG	S	APX	SBZ	AP	C	B	0	1
Extended small page	Extended small page base address			nG	S	APX	TEX	AP	C	B	1	XN

Second-level page table descriptor fields

The fields in a second-level page table have the following meanings:

Bits[1:0]   Identify the type of descriptor (and include XN bit in revised VMSAv6 format).

Bits[3:2]   Are the cacheable and bufferable bits (see *Cacheability and bufferability* on page B2-20).

Bits[11:4]   Are the access permission bits, full page or AP0 subpage.

The following bits are used for the corresponding physical address bits, the field size depending on the page size:

Bits[31:16]   large (64KB) pages

Bits[31:12]   small (4KB) pages

Bits[31:10]   tiny (1KB) pages.

The following bits are used for additional access control functions:

Bits[15:6]   depending on

the format:

- large page control
- subpage access permissions
- TEX
- APX
- S
- nG
- XN

Bits[11:6]   depending on

the format:

- small page control
- subpage access permissions
- TEX
- APX
- S
- nG

Bits[9:6]   tiny page control, SBZ.

For details of these fields see the following sections:

AP and APX   B_TBD_mmu_access

C, B and TEX   B_TBD_mmu_memregion

XN, nG and S   B_TBD_mmu_pagetab_in_v6.

Where subpages are supported, the page is divided into four blocks, each of the same size. AP0 refers to the block with the lowest block base address, with AP1, AP2 and AP3 applying to blocks with incrementing block base addresses.

B4.7.9 Translating page references in coarse page tables

Figure B4-7 shows the complete translation sequence for a 64KB large page in a coarse second-level table.

--- **Note** ---

Because the upper four bits of the Page Index and low-order four bits of the Second-level Table Index overlap, each page table entry for a large page must be repeated 16 times (in consecutive memory locations) in a coarse page table.
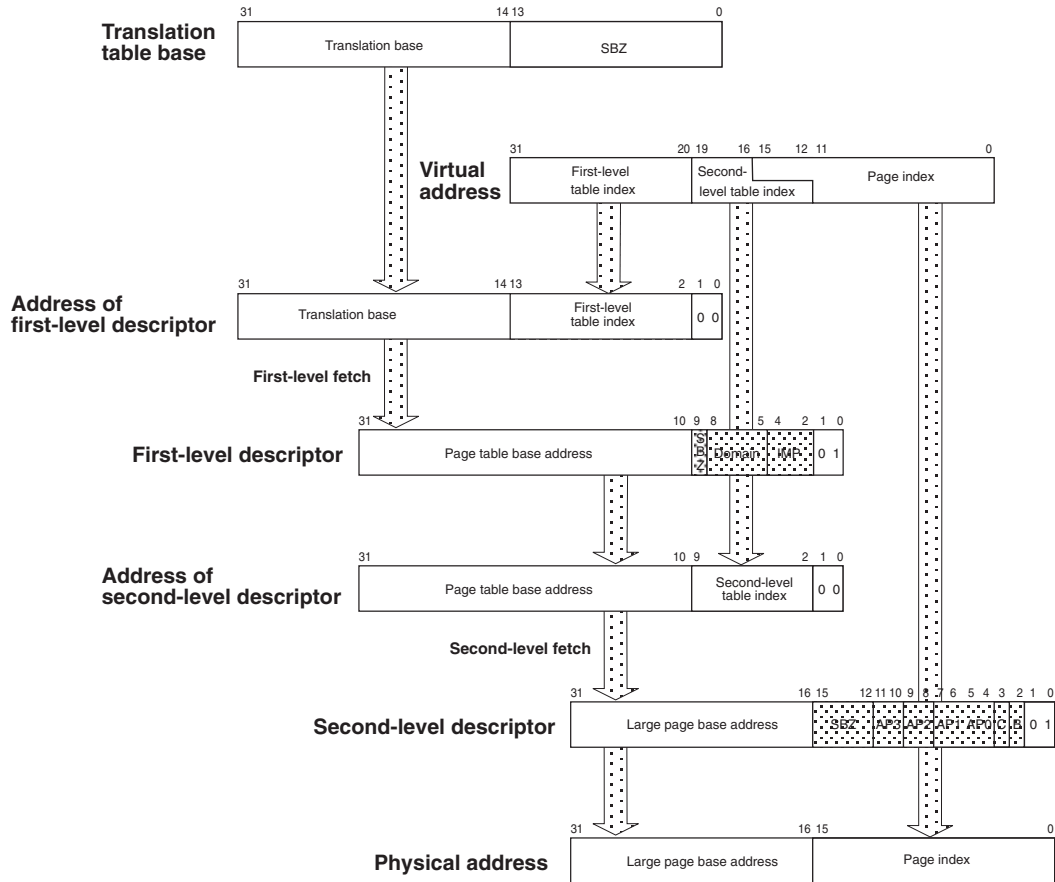
Figure B4-7 Large page translation in a coarse second-level table

ARM DDI 0100F-02

Figure B4-8 shows the complete translation sequence for a 4KB small (standard or extended) page in a coarse second-level table.
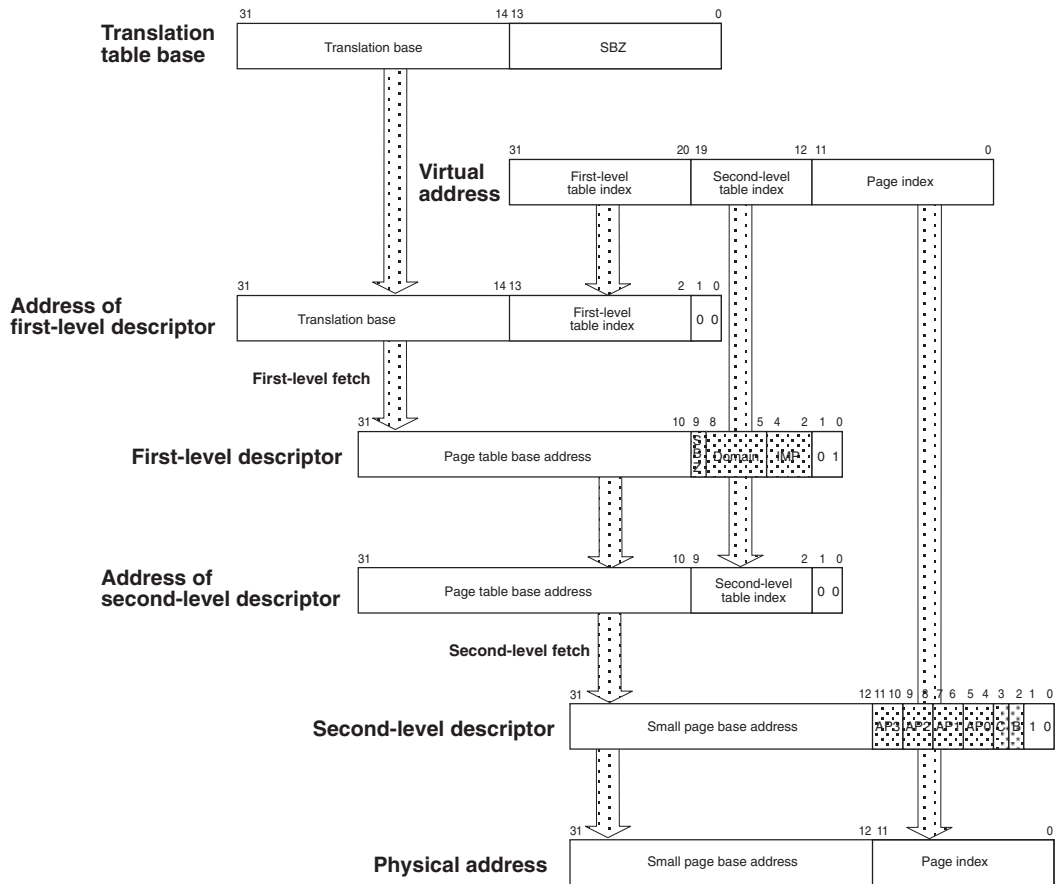
Figure B4-8 Small page translation in a coarse second-level table

Copyright © 1996–2003 ARM Limited. All rights reserved.
Confidential - Draft

B4.7.10   Translating page references in fine page tables (ARMv4 and ARMv5 only)

The translation sequence for a large or small page in a fine second-level table is similar to that for a coarse page, but with the address of the second-level descriptor being determined as shown in Figure B4-9.

When a small page appears in a fine second-level table, the upper two bits of the Page Index and the low-order two bits of the Second-level Table Index overlap; bits[11:10]. Each page table entry for a small page must be repeated four times (in consecutive memory locations) in a fine page table. For a large page the overlap is six bits, bits[15:10], and each page entry must be repeated sixty-four times.

Tiny pages have no overlap, with one entry per 1KB page. Figure TBD_L2transTP shows the complete translation sequence for a 1KB tiny page in a fine second-level table.
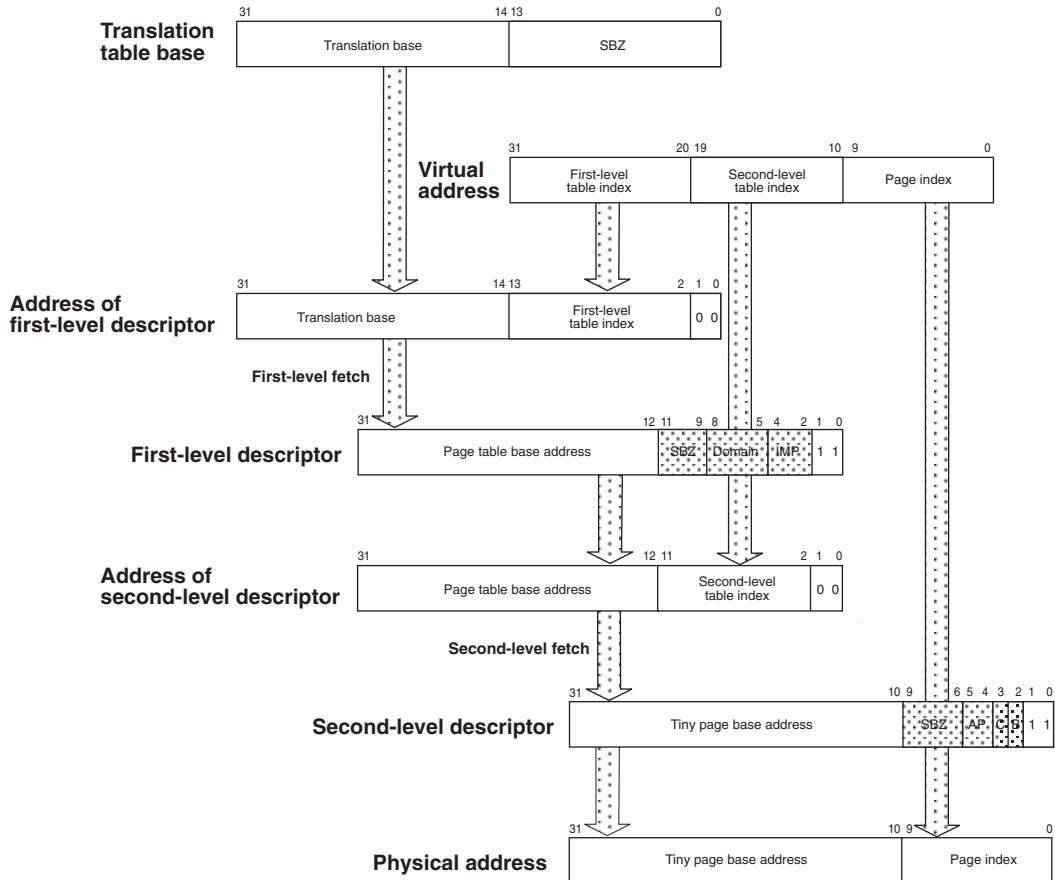
Figure B4-9 Tiny page translation in a fine second-level table

B4.8 CP15 registers

The MMU is controlled with the System Control coprocessor registers. From VMSAv6, several new registers, and register fields have been added:

- a TLB type register in register 0
- additional control bits to register 1
- a second translation table base register, and new control fields to register 2
- an additional fault status register to register 5
- an additional fault address register to register 6
- TLB invalidate by ASID support in register 8
- ASID control in register 13.

Domain support (register 3) and TLB lockdown support (register 10) are the same as in earlier versions of the architecture.

All VMSA-related registers are accessed with instructions of the form:

```
MRC p15, 0, Rd, CRn, CRm, opcode_2
MCR p15, 0, Rd, CRn, CRm, opcode_2
```

Where CRn is the system control coprocessor register. Unless specified otherwise CRm and opcode_2 SBZ.

B4.8.1 Register 0: TLB type register (VMSAv6)

The TLB size and organization is implementation defined. This read-only register describes the number of lockable TLB entries, and whether separate instruction and data or a unified TLB is present. This allows operating systems to establish how to manage the TLB. The TLB type register is accessed by reading CP15 register 0 with the opcode_2 field set to 0b011. For example:

```
MRC p15, 0, Rd, c0, c0, 3 ; returns TLB Type register
```

bit[0]	0 = Unified TLB
	1 = Separate instruction/data TLBs.
Bits[7:0]	SBZ
Bits[15:8]	Number of unified/data TLB lockable entries. $0 <= N <= 255$.
Bits[23:16]	Number of instruction TLB lockable entries. $0 <= N <= 255$.
Bits[31:24]	SBZ

B4.8.2 Register 1: Control register

The following bits in the System Control coprocessor register 1 are used to control the MMU:

M (bit[0])	This is the enable/disable bit for the MMU:
	0 = MMU disabled

---

1 = MMU enabled.

A (bit[1]) This is the enable/disable bit for alignment fault checking (see *Alignment fault* on page B4-15):

0 = Alignment fault checking disabled

1 = Alignment fault checking enabled.

W (bit[3]) This is the enable/disable bit for the write buffer.

0 = write buffer disabled

1 = write buffer enabled

Implementations can choose not to include the W bit. In this case this bit reads as 1 and ignores writes.

S (bit[8]) System protection bit. This feature is deprecated from VMSAv6. The effect of this bit is defined in section B_TBD_mmu_access_permissions.

R (bit[9]) ROM protection bit. This feature is deprecated from VMSAv6. The effect of this bit is defined in section B_TBD_mmu_access_permissions.

XP (bit[23]) Extended page table configuration. This bit configures the hardware page table translation mechanism:

0 = VMSAv5 compatible page table format.

1 = VMSAv6 page table format.

EE (bit[25]) Exception Endian bit, VMSAv6 only. The EE bit is used to define the value of the CPSR E-bit on entry to an exception vector including reset. The value is also used to indicate the endianness of page table data for page table lookups.

B4.8.3   Register 2: Translation table base

Two translation table base registers, and a control register are provided, as shown in Table B4-14:

Table B4-14 Translation table registers

Register name	Opcode2
Translation Table Base 0 (TTBR0)	0
Translation Table Base 1(TTBR0)	1
Translation Table Base Control	2

The translation table base control register determines if a page table miss for a specific virtual address should use translation table base register 0, or translation table base register 1. Its format is:

31	3	2	0
UNP/SBZ		N	
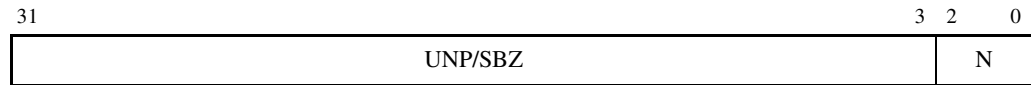
The page table base register is selected as follows:

If N = 0 always use TTBR0. When N = 0 (the reset case), the translation table base is backwards compatible with earlier versions of the architecture.

If N > 0 then if bits [31:32-N] of the virtual address are all zero, use TTBR0, otherwise use TTBR1. N must be in the range 0 <= N <= 7. Therefore for N = 1; if VA[31] == 0, use TTBR0, otherwise use TTBR1. For N = 2; if VA[31:30] == 0b00 use TTBR0, otherwise use TTBR1.

The format for TTBR0 is as follows:

31	14 – n	13 – n	5	4	3	2	1	0
Translation Table Base 0		UNP/SBZ		RGN	P	S	C	
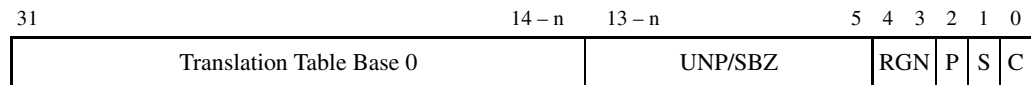
Only bits [31:14-N] of the translation table base 0 register are significant. Therefore if N = 0, the page table must reside on a 16KB boundary, and, for example, if N = 1, it must reside on an 8KB boundary.

The format for TTBR1 is as follows:

31	14	13	5	4	3	2	1	0
Translation Table Base 1		UNP/SBZ		RGN	P	S	C	
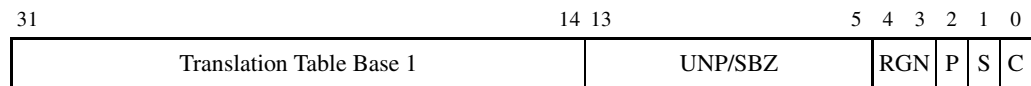
Only bits [31:14] of the translation table base 1 register are significant. Therefore TTBR1 must reside on a 16KB boundary.

The expected use for these two page table base registers is for TTBR1 to be used for operating system and I/O addresses. These do not change on context switches. TTRB0 is used for process specific addresses with each process maintaining a separate first level page table. On a context switch TTBR0 and the ContextID register are modified.

The RGN, P, S, and C bits provide control over the memory attributes for the page table walk:

RGN	Indicates if the page table memory is cacheable beyond level 1 memory:	
	00	VMSAv5: outer noncacheable
		VMSAv6: normal memory noncacheable
	01	UNPREDICTABLE
	10	outer cacheable write-through
	11	outer cacheable write-back.
P	Indicates to the memory controller that if supported ECC is (1) enabled, or (0) disabled.	
S	The page table walk is to shareable (1) or not-shared (0) memory.	
C	Page table walk is inner cacheable (1) or inner non-cacheable (0).	

All page table accesses are outer-cacheable.

───── **Note** ─────

It is IMPLEMENTATION DEFINED whether a page table walk can read from L1 cache. Therefore to ensure coherency, either page tables must be stored in inner write-through memory or, if in inner write-back, the appropriate cache entries must be cleaned after modification to ensure they are seen by the hardware page table walking mechanism.

B4.8.4 Register 3: Domain access control

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

D15	D14	D13	D12	D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0

The Domain Access Control register consists of 16 two-bit fields, each defining the access permissions for one of the 16 domains. Domain values are defined in section B_TBD_mmu_domains.

B4.8.5 Register 4: Reserved

Reading and writing CP15 register 4 is UNPREDICTABLE.

B4.8.6 Register 5: Fault status

This register enables the data and instruction fault status registers to be accessed, depending on the value of the Opcode2 field, as shown in Table B4-15. Prior to VMSAv6, only a combined FSR was defined.

Table B4-15 Fault Status Registers

Register name	Opcode2
Combined/Data FSR	0
Instruction FSR	1

Reading CP15 register 5 returns the value of the Data or Instruction Faults Status Register (DFSR/IFSR). The fault status register contains the source of the last abort. It indicates the domain (when available) and type of access being attempted when an abort occurred.

Bit[11]   Added in VMSAv6. Indicates whether the aborted access was a read or write access. For the IFSR, this bit SBZ.

Bits[7:4]   For the data fault status register only, specifies which domain was being accessed when a memory system abort occurred.

Bits[10, 3:0]   The reason for the abort.

ARM DDI 0100F-02

Data Fault Status Register

31	12	11	10	9	8	7	4	3	0
UNP/SBZ		W R	FS [4]	UNP/ SBZ	0	Domain		Status	

Instruction Fault Status Register

31	11	10	9	4	3	0
UNP/SBZ		FS [4]	UNP/SBZ		Status	
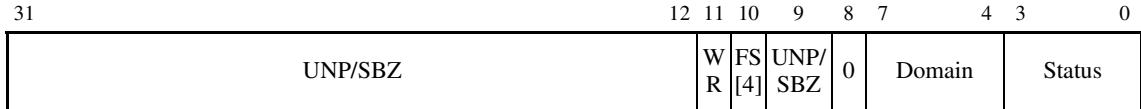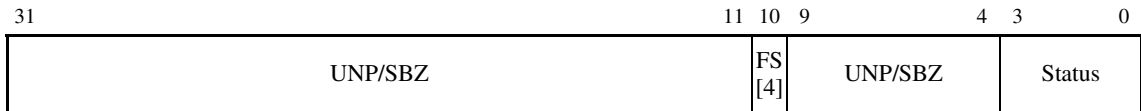
B4.8.7   Register 6: Fault Address Register

This register enables the data and watchpoint fault address registers to be accessed, depending on the value of the Opcode2 field, as shown in Table B4-16. Prior to VMSAv6, only a combined FAR was defined.

Table B4-16 Fault Address registers

Register name	Opcode2
Combined/Data FAR	0
Watchpoint FAR (WFAR)	1

The FAR and WFAR are updated on an abort in accordance with Table B4-7 on page B4-21.

———— **Note** ————

The contents of the WFAR are a virtual address, not a *Modified Virtual Address* (MVA). The FAR contains a MVA where the FCSE mechanism is in use (see chapter B_TBD_FCSE).

———————————

Writing CP15 register 6 enables the values of the FAR and WFAR to be written. This is useful for a debugger to restore their values. When the FAR is written by an MCR instruction, its value is treated as data, and no address modification is performed by the FCSE.

B4.8.8   Register 8: TLB functions

CP15 register 8 is a write-only register that is used to control TLBs. Table B4-17 shows the defined TLB operations and the values of `<CRm>` and `<opcode2>` used in the `MCR` instruction for each of them. The results of using any combination of `<CRm>` and `<opcode2>` not specified in the table are UNPREDICTABLE.

Functions that update the contents of the TLB occur in program order. Therefore an explicit data access prior to the TLB function uses the old TLB contents, and an explicit data access after the TLB function uses the new TLB contents. For instruction accesses, prefetching and branch prediction can mean instructions following the TLB operation are already in the process of being fetched and scheduled for execution, requiring special care to ensure the expected instruction execution behavior. TLB updates are guaranteed to have taken effect by the next pipeline flush. This includes PrefetchFlush operations and exception return sequences.

Attempting to read CP15 register 8 with an `MRC` instruction is UNPREDICTABLE.

Table B4-17 TLB functions

Function	Data	Instruction
Invalidate entire unified TLB or both instruction and data TLBs	SBZ	`MCR p15, 0, Rd, c8, c7, 0`
Invalidate single entry	Modified virtual address	`MCR p15, 0, Rd, c8, c7, 1`
Invalidate on ASID match unified TLB	-	`MCR p15, 0, Rd, c8, c7, 2`
Invalidate entire instruction TLB	SBZ	`MCR p15, 0, Rd, c8, c5, 0`
Invalidate instruction single entry	Modified virtual address	`MCR p15, 0, Rd, c8, c5, 1`
Invalidate on ASID match instruction TLB	-	`MCR p15, 0, Rd, c8, c5, 2`
Invalidate entire data TLB	SBZ	`MCR p15, 0, Rd, c8, c6, 0`
Invalidate data single entry	Modified virtual address	`MCR p15, 0, Rd, c8, c6, 1`
Invalidate on ASID match data TLB	-	`MCR p15, 0, Rd, c8, c6, 2`

If the instruction or data TLB operations are used on an implementation with a unified TLB, the function is performed on the unified TLB.

Invalidate TLB

This invalidates all unlocked entries in the TLB. This function causes the prefetch buffer to be flushed. This ensures that, following the TLB invalidation, all following instructions are fetched using the page tables in memory.

Invalidate Single Entry

Invalidate single entry can be used to invalidate an area of memory prior to remapping. For each area of memory to be remapped (section, tiny page pre-VMSAv6, small page, or large page) an invalidate single entry of a virtual address in that area should be performed.

This function invalidates a TLB entry that matches the provided virtual address and ASID, or a global TLB entry that matches the provided VA.

If the Page Table VA register Process field selects global entries then this function has no effect.

Invalidate on ASID Match

This is a single interruptible operation that invalidates all TLB entries which match the provided ASID. This function invalidates locked entries.

In implementations with set-associative TLBs, this operation can take a number of cycles to complete and the instruction can be interruptible. When interrupted the R14 state is such as to indicate that the MCR instruction had not executed. Therefore R14 points to the address of the MCR + 4, The interrupt routine automatically restarts at the MCR instruction.

If the instruction or TLB operations are used on an implementation with a unified TLB, the equivalent function is performed on the unified TLB.

If this operation is interrupted and later restarted, it is UNPREDICTABLE whether any entries fetched into the TLB by the interrupt that use the provided ASID are invalidated by the restarted invalidation.

The Invalidate Single Entry function requires a virtual address as an argument and, in VMSAv6 with the XP control bit set, an ASID. The ASID is not checked for global TLB entries.

Where ASIDs are not supported, a 32-bit address can be provided. Where an ASID is required, the format for the virtual address is as follows:

31	12 11	8 7	0
MVA	SBZ	ASID	
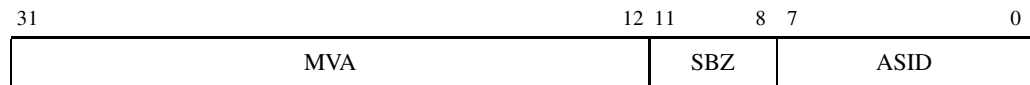
Invalidate on ASID Match requires an ASID as an argument. The format is as follows:

31	8 7	0
SBZ	ASID	
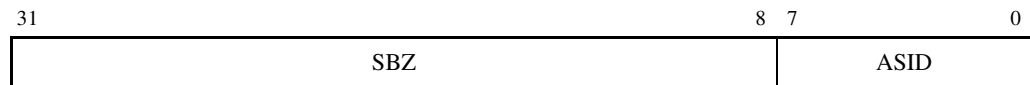
B4.8.9   Register 10: TLB lockdown

TLB lockdown is a feature of some ARM memory systems that allows the results of specified translation table walks to be loaded into the TLB, in such a way that they are not overwritten by the results of subsequent translation table walks. It is programmed via CP15 register 10.

Translation table walks can take a considerable amount of time, especially as they involve potentially slow main memory accesses. In real-time interrupt handlers, translation table walks caused by the TLB not containing translations for the handler and/or the data it accesses can increase interrupt latency significantly.

TLB lockdown is a feature of some ARM memory systems which allows the results of specified translation table walks to be loaded into the TLB, in such a way that they are not overwritten by the results of subsequent translation table walks. It is programmed via CP15 register 10.

If W is the logarithm base 2 of the number of TLB entries, rounded up to an integer if necessary, then the format of CP15 register 10 is:

31	32-W 31-W	32-2W 31-2W	1 0
base	victim	UNP/SBZP	P

If the implementation has separate instruction and data TLBs, there are two variants of this register, selected by the `<opcode2>` field of the `MCR` or `MRC` instruction used to access register 10:

`<opcode2> == 0`   Selects the data TLB lockdown register.

`<opcode2> == 1`   Selects the instruction TLB lockdown register.

If the implementation has a unified TLB, only one variant of this register exists, and `<opcode2>` SBZ.

`<CRm>` must always be c0 for `MCR` and `MRC` instructions that access register 10.

Writing register 10 has the following effects:

- The victim field specifies which TLB entry is replaced by the translation table walk result generated by the next TLB miss.

- The base field constrains the TLB replacement strategy to only use the TLB entries numbered from (base) to (number of TLB entries)-1, provided the victim field is already in that range.

- Any translation table walk results written to TLB entries while P == 1 are protected from being invalidated by the register 8 invalidate entire TLB operations. Ones written while P == 0 are invalidated normally by these operations.

——— **Note** ———

If the number of TLB entries is not a power of two, writing a value to either the base or victim fields which is greater than or equal to the number of TLB entries has UNPREDICTABLE results.

Reading register 10 returns the last values written to the base field and the P bit, and the number of the next TLB entry to be replaced in the victim field.

TLB lockdown procedure

The normal procedure to lock down N TLB entries is as follows:

1. Ensure that no processor exceptions can occur during the execution of this procedure, by disabling interrupts, and so on.

2.   If an instruction TLB or unified TLB is being locked down, write the appropriate version of register 10 with base == N, index == N, and P == 0. If appropriate, also turn off facilities like branch prediction that make instruction prefetching harder to understand.

3.   Invalidate the entire TLB to be locked down.

4.   If an instruction TLB is being locked down, ensure that all TLB entries are loaded which relate to any instruction that could be prefetched by the rest of the lockdown procedure. (Provided care is taken about where the lockdown procedure starts, it is normally possible for one TLB entry to cover all of these, in which case the first instruction prefetch after the TLB is invalidated can do this job.)

      If a data TLB is being locked down, ensure that all TLB entries are loaded which relate to any data accessed by the rest of the lockdown procedure, including any inline literals used by its code. (This is usually best done by avoiding the use of inline literals in the lockdown procedure and by putting all other data used by it in an area covered by a single TLB entry, then loading one data item.)

      If a unified TLB is being locked down, do both of the above.

5.   For each of i = 0 to N-1:

      a.   Write to register 10 with base == i, victim == i, and P == 1.

      b.   Force a translation table walk to occur for the area of memory whose translation table walk result is to be locked into TLB entry i, by:

            •   If a data TLB or unified TLB is being locked down, loading an item of data from the area of memory.

            •   If an instruction TLB is being locked down, using the register 7 *prefetch instruction cache line* operation defined in *Register 7: cache management functions* on page B6-18 to cause an instruction to be prefetched from the area of memory.

6.   Write to register 10 with base == N, victim == N, and P == 0.

───── **Note** ─────

If the *Fast Context Switch Extension* (FCSE) (see Chapter B8), is being used, care is required in step 5b, because:

•   If a data TLB or a unified TLB is being locked down, the address used for the load instruction is subject to modification by the FCSE.

•   If an instruction TLB is being locked down, the address used for the register 7 operation is being treated as data and so is not subject to modification by the FCSE.

To minimize the possible confusion caused by this, it is recommended that the lockdown procedure should:

•   start by disabling the FCSE (by setting the PID to zero)

•   where appropriate, generate modified virtual addresses itself by ORing the appropriate PID value into the top 7 bits of the virtual addresses it uses.

TLB unlock procedure

To unlock the locked-down portion of the TLB after it has been locked down using the above procedure:

1.   Use register 8 operations to invalidate each single entry that was locked down.

2.   Write to register 10 with base == 0, victim == 0, and P == 0.

—— **Note** ——

Step 1 is used to ensure that P == 1 entries are not left in the TLB. If they were left in the TLB, the entire TLB invalidation step (step 3) of a subsequent TLB lockdown procedure would not have the desired effect.

B4.8.10 Register 13: Process ID

This register determines the currently running process. Two different values can be stored depending on the opcode2 field, see Table B4-18. When updating the ASID the current instruction and data stream should be in a global, not an ASID dependent memory region. On reset the values of both these registers SBZ.

Table B4-18 Process ID registers

Register name	Opcode2
FCSE PID	0
ContextID	1

FCSE PID

Controls the *Fast Context Switch Extension* (FCSE). The use of the FCSE is deprecated.

31	25 24	0
FCSE PID	UNP/SBZP	

ContextID

The bottom eight bits of this register are the currently running ASID. The top bits extend the ASID into a general-purpose process ID.

Implementations can make this value available to the rest of the system. To ensure that all accesses are related to the correct ContextID, software should execute a Drain Write Buffer operation before changing this register.

The whole of this register is used by both the *Embedded Trace Macrocell* (ETM) and by the debug logic. Its value can be broadcast by the ETM to indicate the currently running process and should be programmed with a unique number for each process. Therefore if an ASID is reused the ETM can distinguish between processes. It is used by ETM to determine how virtual to physically memory is mapped.

ARM DDI 0100F-02

Its value can also be used to enable process-dependent breakpoints and instructions.

After changing this register an IMB sequence needs to be executed before any instructions are executed that are from an ASID dependent memory region. Code that updates the ASID must be executed from a global memory region.

31	25 24	8	7	0
	PROCID		ASID	
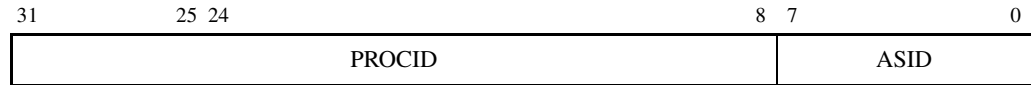
Chapter B5
Protected Memory System Architecture

This chapter describes the ARM Protection Unit. It contains the following sections:

*   *About the Protection Unit* on page B5-2
*   *Overlapping regions* on page B5-5
*   *CP15 registers* on page B5-6.

——— **TBD** ———

NOTE: This chapter has not yet been updated to reflect PMSAv6.

B5.1 About the Protection Unit

The *Protection Unit* memory system architecture provides a considerably simpler alternative to the MMU memory system architecture described in Chapter B4 *Virtual Memory System Architecture*. This allows both hardware and software to be simplified in systems which do not require all facilities of the MMU.

The main simplification is that the Protection Unit does not use translation tables. Instead, up to eight *protection regions* can be defined by writing to Protection Unit registers. This eliminates the need for hardware to do translation table walks, and for software to set up and maintain the translation tables. The cost of this simplification is that the level of control over memory behavior is considerably less fine-grained. MMU translation tables can define many thousands of memory regions with different attributes, considerably more than are possible with the Protection Unit.

A second simplification is that virtual-to-physical address mapping is not supported. The physical memory address is always the same as the virtual address generated by the ARM processor (except that if the FCSE is used, as described in TBD xref, the physical address is the same as the *modified virtual address* that it generates).

B5.1.1 Protection regions

The sizes and starting addresses of the eight protection regions are defined using CP15 register 6. The size of a region must be a power of two, and can range from 4KB up to 4GB. The starting address of a region must be a multiple of its size. For example, a 4KB region (size `0x1000`) can start at address `0x12345000`, but an 8KB region (size `0x2000`) or larger can not.

The cacheable bits (C), bufferable bits (B) and *Access Permission* (AP) bits of the eight protection regions are defined by CP15 registers 2, 3 and 5 respectively. These behave similarly to the corresponding bits in the MMU memory system architecture, except that the interpretation of the AP bits is not modified by the System (S) and ROM (R) bits in CP15 register 1. (These bits are therefore not normally implemented in Protection Unit systems.)

Protection regions can legitimately overlap each other. If an address lies within more than one protection region, the attributes of the highest-numbered protection region apply to it. For more details, see *Overlapping regions* on page B5-5.

Some implementations (typically those with separate instruction and data caches) provide two separate sets of eight protection regions, with one set being used for instruction fetches and the other set for data accesses.

———— **Note** ————

It is possible to program the instruction and data protection regions differently in these implementations, but some care must be taken. In particular, most ARM and Thumb code areas include *inline literals*, which are constant addresses and other data items held in the same area as the code itself. Inline literals are automatically generated by compilers, and can also appear in assembler code. (For ARM Development Systems, they are typically generated by the use of `DCD`, `DCB` and related assembler directives, or by the use of the construction `LDR Rn,=expression` together with an `LTORG` directive.) Inline literals are typically accessed by `LDR` instructions with base register R15. Such instructions generate a data access to the code area, so it does not normally make sense to program an instruction protection region without programming a corresponding data protection region to allow the code to access its inline literals.

B5.1.2   Memory access sequence

When the ARM processor generates a memory access, the Protection Unit checks whether the address lies within the range of each of the eight protection regions (using the appropriate set of eight regions if the implementation has separate sets for instruction and data accesses):

*   If the address does not lie in any protection region, the memory access is aborted.

*   If the address lies in one or more protection regions, the access is controlled by the highest-numbered of those regions. The C, B and AP bits associated with that region are used by the access as follows:

    1.   The C (cacheable) and B (bufferable) bits are used to control the cache and write buffer and to determine whether the access is cached or uncached, as described in *Cacheability and bufferability* on page B2-20. (If the system does not contain a cache and/or a write buffer, the relevant bits are ignored.)

    2.   The AP (access permission) bits are used to determine whether the access is permitted. If the access is not permitted, the Protection Unit signals a memory abort to the ARM processor. Otherwise, the access is allowed to proceed.

        Register 5: Access permission bits on page B5-7 and *Exceptions* on page A2-15 describe this in more detail.

Figure B5-1 shows the memory access sequence for a cached system:
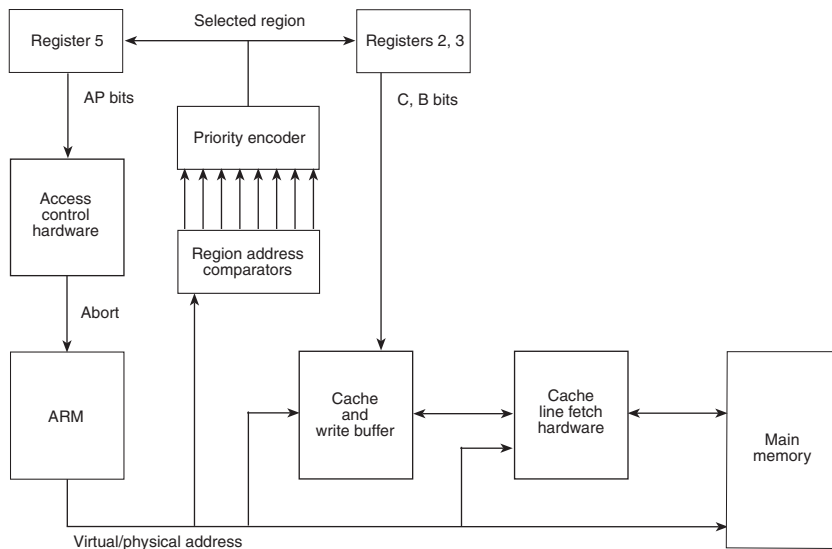
Figure B5-1 Cached Protection Unit memory system overview

B5.1.3 Enabling the Protection Unit

The Protection Unit can be enabled by writing 1 to CP15 register 1 bit[0], and can be disabled by writing 0 to the same bit. On reset, this bit is cleared and all protection regions are invalidated.

Before the Protection Unit is enabled, at least one valid protection region must be specified and its C, B and AP bits must be set. In the case of a Protection Unit with separate sets of protection regions for instruction and data accesses, at least one valid protection region in each of the two sets must be programmed in this manner.

Furthermore, the code that enables the Protection Unit must lie within a valid (instruction) protection region. Enabling the Protection Unit without adhering to these rules can result in behavior which is at best UNPREDICTABLE, and can even result in a hung state which can only be recovered from by resetting the system.

While the Protection Unit is disabled, all memory accesses are treated as uncacheable, unbufferable and non-aborting.

B5.2   Overlapping regions

The Protection Unit can be programmed with two or more overlapping regions. When overlapping regions are programmed, a fixed priority scheme is applied to determine the region whose attributes are applied to the memory access.

Attributes for region 7 take highest priority and those for region 0 take lowest priority. For example:

*   Data region 2 is programmed to be 4KB in size, starting from address `0x3000` with AP == 0b10 (Privileged mode full access, User mode read only).

*   Data region 1 is programmed to be 16KB in size, starting from address `0x0` with AP == 0b01 (Privileged mode access only).

When the processor performs a data load from address `0x3010` while in User mode, the address falls into both region 1 and region 2, as shown in Figure B5-2. Because there is a clash, the attributes associated with region 2 are applied. In this case, the load would not abort.
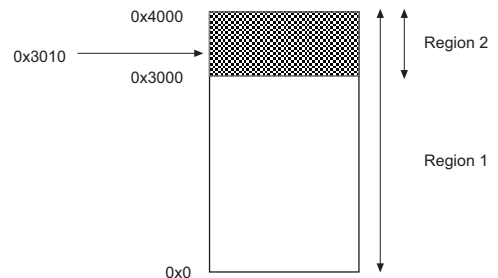
Figure B5-2 Overlapping memory regions

B5.2.1   Background regions

Overlapping regions increase the flexibility of how the eight regions can be mapped onto physical memory devices in the system. The overlapping properties can also be used to specify a background region. For example, there might be a number of physical memory areas sparsely distributed across the 4GB address space. If a programming error occurs therefore, it might be possible for the processor to issue an address which does not fall into any defined region.

If the address issued by the processor falls outside of any of the defined regions, the Protection Unit is hard-wired to abort the access. This behavior can be overridden by programming region 0 to be a 4GB background region. In this way, if the address does not fall into any of the other seven regions, the access is controlled by the attributes the user has specified for region 0.

B5.3   CP15 registers

The Protection Unit is controlled with the System Control coprocessor registers 2, 3, 5, and 6, and one or two bits of register 1.

B5.3.1   Protection Unit control bits in register 1

M(bit[0]) in the System Control coprocessor register 1 is the enable/disable bit for the Protection Unit:

0 = Protection Unit disabled

1 = Protection Unit enabled.

On systems without a Protection Unit or MMU, this bit must read as zero and ignore writes.

A Protection Unit memory system can optionally implement alignment checking, as described in *Alignment fault* on page B4-15. If it does, the A bit (bit[1]) of register 1 behaves as described there and in *Register 0: TLB type register (VMSAv6)* on page B4-37. Otherwise, the A bit SBZ.

B5.3.2   Register 2: Cacheability bits

31				8	7	6	5	4	3	2	1	0
	UNP/SBZP				C7	C6	C5	C4	C3	C2	C1	C0
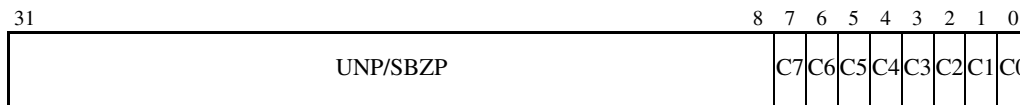
Reading from CP15 register 2 returns the cacheable (C) bits for the eight protection regions in bits[7:0], with bit[n] corresponding to region n, and an UNPREDICTABLE value in bits[31:8].

Writing to CP15 register 2 updates the cacheable (C) bits of the eight protection regions, with the C bit of region n being set to bit[n] of the value written. Bits[31:8] must be written as zero or as a value previously read from bits[31:8] of this register.

In each case, the `<CRm>` field of the `MRC` or `MCR` instruction is ignored and must be c0. If the implementation only has one set of eight protection regions, the `<opcode2>` field should be zero. If it has separate sets of protection regions for instruction and data accesses, `<opcode2>` must be specified as 0 to select the data protection regions and 1 to select the instruction protection regions.

For details of how the C bit affects cache and write buffer behavior, see *Cacheability and bufferability* on page B2-20.

B5.3.3   Register 3: Bufferability bits

31				8	7	6	5	4	3	2	1	0
	UNP/SBZP				B7	B6	B5	B4	B3	B2	B1	B0
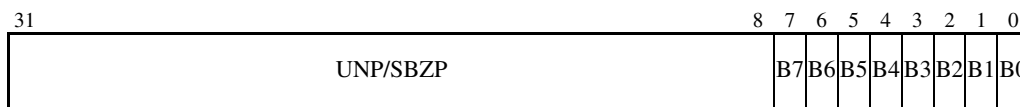
Reading from CP15 register 3 returns the bufferable (B) bits for the eight protection regions in bits[7:0], with bit[n] corresponding to region n, and an UNPREDICTABLE value in bits[31:8].

Writing to CP15 register 3 updates the bufferable (B) bits of the eight regions, with the B bit of region n being set to bit[n] of the value written. Bits[31:8] must be written as zero or as a value previously read from bits[31:8] of this register.

In each case, the <CRm> and <opcode2> fields of the MRC or MCR instruction are ignored and must be c0 and zero respectively.

For details of how the B bit affects cache and write buffer behavior, see *Cacheability and bufferability* on page B2-20.

B5.3.4 Registers 4, 8, 10: Reserved

Accessing (reading or writing) any of these registers is UNPREDICTABLE.

B5.3.5 Register 5: Access permission bits

31	16 15 14	13 12	11 10	9 8	7 6	5 4	3 2	1 0
UNP/SBZP	AP7	AP6	AP5	AP4	AP3	AP2	AP1	AP0

Reading from CP15 register 3 returns the AP bits for the eight protection regions in bits[15:0], with bits[2n+1:2n] corresponding to region n, and an UNPREDICTABLE value in bits[31:16].

Writing to CP15 register 3 updates the AP bits of the eight regions, with the AP bits of region n being set to bits[2n+1:2n] of the value written. Bits[31:16] must be written as zero or as a value previously read from bits[31:16] of this register.

In each case, the <CRm> field of the MRC or MCR instruction is ignored and must be c0. If the implementation only has one set of eight protection regions, the <opcode2> field should be zero. If it has separate sets of protection regions for instruction and data accesses, <opcode2> must be specified as 0 to select the data protection regions and 1 to select the instruction protection regions.

The interpretation of each set of AP bits is as shown in Table B5-1. If the requested type of access is not permitted, an abort is signaled to the ARM processor.

Table B5-1 Protection Unit access permissions

AP	Privileged permissions	User permissions
0b00	No access	No access
0b01	Read/write	No access
0b10	Read/write	Read only
0b11	Read/write	Read/write

B5.3.6   Register 6: Protection area control

31	12	11	6	5	1	0
Base address		UNP/SBZP		Size		E
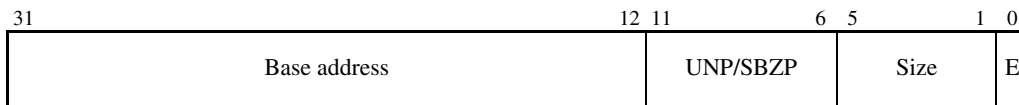
Reading from CP15 register 6 returns the current base address, size and enabled/disabled status of a protection region, in the format shown in the above diagram. The value read for bits[11:6] is UNPREDICTABLE.

Writing to CP15 register 6 sets the base address, size and enabled/disabled status of a protection region, in the format shown in the above diagram. The value written to bits[11:6] must either be zero or a value previously read from bits[11:6] of CP15 register 6.

There is one version of register 6 for each protection region in the Protection Unit. The version used (and therefore the protection region affected) is selected by the <CRm> and <opcode2> fields of the MCR or MRC instruction used to access the register:

•   <CRm> is used to select the number of the protection region, by specifying c0 to select protection region 0, c1 to select protection region 1, and so on, through to c7 to select protection region 7.

•   If the implementation only has one set of eight protection regions, the <opcode2> field should be zero.

•   If the implementation has separate sets of protection regions for instruction and data accesses, <opcode2> must be specified as 0 to select a data protection region and 1 to select an instruction protection region.

The meaning of the fields in the value read from or written to register 6 is as follows:

•   The E bit enables or disables the associated protection region:

0 = protection region disabled

1 = protection region enabled.

A disabled protection region never matches any addresses, and therefore does not affect the memory access sequence in any way. All protection regions are disabled on reset.

•   The Size field selects the associated protection region's size, which can vary from 4KB to 4GB. The encoding is shown in Table B5-2 on page B5-9.

•   The Base address field specifies bits[31:12] of the address of the first byte in the associated protection region.

The address of this first byte is required to be a multiple of the region size, so its bits[11:0] are always zero and do not need to be programmed. Also, because of this requirement for the region to be aligned to a multiple of its size, one or more further bits of the base address might also be constrained to be zero. These additional constraints are shown in Table B5-2 on page B5-9. If they are broken, the protection region is misaligned, and UNPREDICTABLE behavior might result.

Table B5-2 Region size encoding

Size field	Area size	Base area constraints
0b00000-0b01010	UNPREDICTABLE	-
0b01011	4KB	None
0b01100	8KB	Bit[12] must be zero
0b01101	16KB	Bits[13:12] must be zero
0b01110	32KB	Bits[14:12] must be zero
0b01111	64KB	Bits[15:12] must be zero
0b10000	128KB	Bits[16:12] must be zero
0b10001	256KB	Bits[17:12] must be zero
0b10010	512KB	Bits[18:12] must be zero
0b10011	1MB	Bits[19:12] must be zero
0b10100	2MB	Bits[20:12] must be zero
0b10101	4MB	Bits[21:12] must be zero
0b10110	8MB	Bits[22:12] must be zero
0b10111	16MB	Bits[23:12] must be zero
0b11000	32MB	Bits[24:12] must be zero
0b11001	64MB	Bits[25:12] must be zero
0b11010	128MB	Bits[26:12] must be zero
0b11011	256MB	Bits[27:12] must be zero
0b11100	512MB	Bits[28:12] must be zero
0b11101	1GB	Bits[29:12] must be zero
0b11110	2GB	Bits[30:12] must be zero
0b11111	4GB	Bits[31:12] must be zero

Chapter B6
Caches and Write Buffers

This chapter describes cache and write buffer control functions that are common to both the MMU-based memory system and the Protection Unit-based memory system. It contains the following sections:

Prior to ARMv6, architecture guidelines were provided using the System Control Coprocessor (CP15) for configuration and management functions. ARMv6 mandates use of the System Control Coprocessor, and has extended the provisions of earlier architecture variants. New features introduced with ARMv6 are marked where appropriate. It is IMPLEMENTATION DEPENDENT whether these are adopted by implementations compliant to earlier (ARMv4 or ARMv5) versions of the architecture.

B6.1   About caches and write buffers

Caches and write buffers to improve average system performance are now commonplace in ARM memory systems. Core clock rates have increased at a faster rate than memory access times over recent years. This factor, and smaller process geometries, the economics of on-chip memory, and system power constraints have encouraged the usage of caches to meet growing system demands. However, the relative cost of closely-coupled memory over memory at other points in the hierarchy (see section B_TBD_mem_hierarchy) remains high. Therefore closely coupled-memory is well suited to the shared use memory model caches and write buffers provide.

A cache is a block of high-speed memory locations containing both address information (commonly known as TAG bits) and the associated data,. The purpose is to increase the average speed of a memory access. Caches operate on two principles of locality:

Spatial locality   an access to one location is likely to be followed by accesses from adjacent locations, for example sequential instruction execution or usage of a data structure

Temporal locality   an access to an area of memory is likely to be repeated within a short time period, for example execution of a code loop.

To minimize the quantity of control information stored, the spatial locality property is used to group several locations together under the same TAG. This logical block of memory locations is commonly known as a cache line, and is typically 32 bytes long. When data is loaded into a cache, access times for subsequent loads and stores are dramatically reduced, resulting in overall performance benefits. An access to information already in the cache is known as a cache hit, and other accesses are called cache misses. Systems separate memory accesses into cacheable and non-cacheable regions. The reasons why not all accesses can be cacheable is explained TBD xref elsewhere in this chapter.

Normally, caches are self-managing with the updates occurring automatically. Whenever the processor wishes to accesses a cacheable location, the cache is checked. If the access is a cache hit, the access occurs immediately, otherwise a location is allocated and the cache line loaded from memory. Different cache topologies and access policies are possible. All cache topologies and access policies must comply with a memory coherence model. See Chapter B2 *Memory Order Model* for more details on memory ordering.

A write buffer is a block of high-speed memory whose purpose is to optimize stores to main memory. When a store occurs, its data, address and other details, for example data size, are written to the write buffer at high speed. The write buffer then completes the store at main memory speed. This is typically much slower than the speed of the ARM processor. In the meantime, the ARM processor can proceed to execute further instructions at full speed.

Write buffers and caches introduce a number of potential problems, mainly because of:

* memory accesses occurring at times other than when the programmer would normally expect them
* there being multiple physical locations where a data item can be held.

This chapter discusses cache features, associated problems, and the cache and write buffer control facilities that can be used to manage them. They are common to the MMU system architecture described in Chapter B4 *Virtual Memory System Architecture* and the PMU system architecture described in Chapter B5 *Protected Memory System Architecture*.

Prior to ARMv6, caches associated with ARM cores traditionally used virtual addressing. This implies that they need to be invalidated or cleaned or both when the virtual-to-physical address mapping changes or in certain other circumstances, as described in B_TBD_cache_mem_coherency. ARMv6 introduces cache behavior typically associated with physical caches, designed to reduce the need for flushing entries on context switches.

If the *Fast Context Switch Extension* (FCSE) described in chapter B_FCSE is being used, all references to virtual addresses in this chapter mean the modified virtual address that it generates.

————— **Note** —————

Use of the FCSE is deprecated in ARMv6.

B6.2   Cache organization

The basic unit of storage in a cache is the cache line. A cache line is said to be *valid* when it contains cached data or instructions, and *invalid* when it does not. All cache lines in a cache are invalidated on reset. A cache line becomes valid when data or instructions are loaded into it from memory.

When a cache line is valid, it contains up-to-date values for a block of consecutive main memory locations. The length of a cache line is always a power of two, and is typically 16 bytes (4 words) or 32 bytes (8 words). If the cache line length is 2^L bytes, the block of main memory locations is always 2^L-byte aligned. Because of this alignment requirement, virtual address bits[31:L] are identical for all bytes in a cache line.

A cache hit occurs when bits[31:L] of the address supplied by the ARM processor match the same bits of the address associated with a valid cache line. Traditionally this has been a virtual address match. However, from ARMv6 the prescribed behavior is aligned to a physically addressed cache, see section B_TBD_L1_overview_v6.

A cache is usually divided into a number of cache sets, with a fixed number of cache lines associated with each set. The number of cache sets (NSETS) is always a power of two. If the cache line length is 2^L bytes and there are 2^S cache sets, bits[L+S-1:L] of the virtual address supplied by the ARM processor are used to select a cache set. Only the cache lines in that set are allowed to hold the data or instructions at the address. These are typically checked in parallel for performance reasons.

The remaining bits of the virtual address (bits[31:L+S]) are known as the *tag bits*. A cache hit occurs if the tag bits of the address supplied by the ARM processor match the tag bits associated with a valid line in the selected cache set. Caches that follow this topology are often referred to as *N-way set-associative caches*.

Figure B6-1 shows how the virtual address is used to look up data or instructions in the cache.
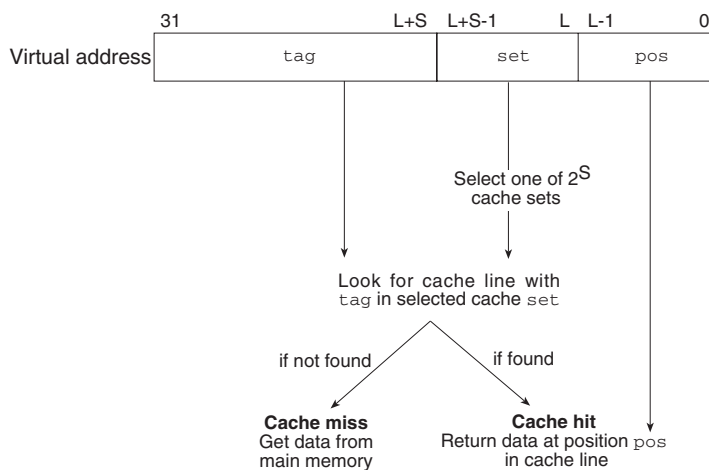
Figure B6-1 Cache look-up

ARM DDI 0100F-02

B6.2.1   Set-associativity

The *set-associativity* of a cache is the number of cache lines in each of its cache sets, and is referred to as the ASSOCIATIVITY or the number of cache ways (NWAYS). It can be any number ≥ 1, and is not restricted to being a power of two.

Low set-associativity generally simplifies cache look-up. However, if the number of frequently-used memory cache lines that use a particular cache set exceeds the set-associativity, main memory activity goes up and performance drops. This is known as *cache contention*, and becomes more likely as the set associativity is decreased.

The two extreme cases are fully associative caches and direct-mapped caches:

*   A *fully associative* cache has just one cache set, that consists of the entire cache. It is N-way set-associative, where N is the total number of cache lines in the cache. Any cache look-up in a fully associative cache must check every cache line.

*   A *direct-mapped* cache is a one-way set-associative cache. Each cache set consists of a single cache line, so cache look-up must select and check only one cache line. However, cache contention is particularly likely to occur in direct-mapped caches.

Within each cache set, the cache lines are numbered from 0 to (set associativity)-1. The number associated with each cache line is known as its *way number*. The way number, together with the set address field, identifies a specific cache line block in the cache memory.

A *cache way* is defined as all the (2^S) cache lines associated with a specific value of the way number. Some cache operations take a cache way number as a parameter, to allow a software loop to work systematically through a cache set, for example the cache lockdown mechanism described in section B_TBD_CWB_cache_lockdown.

———— **Note** ————

In previous versions of the ARM architecture, the value associated with the cache way has been referred to as the *index*. However, computer architecture convention is to associate usage of the word index with the cache set address field. To remove this ambiguity, the terms SET and WAY are used here. Therefore cache invalidate operations on a specific cache line will be described as *invalidate by SET/WAY* rather than *invalidate by SET/INDEX*. Any use of the word *index* in a cache context relates to the SET convention, that is, a virtually-indexed cache. Ambiguity might exist in implementation documentation. Care is required when correlating information between this architecture manual and implementation data, for example technical reference manuals.

B6.2.2   Cache size

Generally, as the size of a cache increases, a higher percentage of memory accesses are cache hits. This reduces the average time per memory access and so improves performance. However, a large cache typically uses a significant amount of silicon area and power. Different sizes of cache can therefore be used in an ARM memory system, depending on the relative importance of performance, silicon area, and power consumption.

The cache size can be broken down into a product of three factors:

•   The cache line length LINELEN, measured in bytes.

•   The set-associativity ASSOCIATIVITY. A cache set consists of ASSOCIATIVITY cache lines, so the size of a cache set is ASSOCIATIVITY $\times$ LINELEN.

•   The number NSETS of cache sets making up the cache.

Using the sizing and address bit definitions defined in section B_TBD_cache_org:

```
Cache size = ASSOCIATIVITY × NSETS × LINELEN
           = NWAYS  × NSETS  × LINELEN
           = NWAYS  × 2^S  × 2^L bytes
```

If separate data and instruction caches are used, different values of these parameters can be used for each, and the resulting cache sizes can be different.

From ARMv6, the System Control Coprocessor Cache Type Register is the mandated method to define the L1 caches, see section B_CP15_cache_type_register. It is also the recommended method for earlier variants of the architecture. In addition, sections B_TBD_cache_L2_provision and B_CP15_L2cache describe architecture guidelines for level 2 cache support.

Copyright © 1996–2003 ARM Limited. All rights reserved.   ARM DDI 0100F-02
Confidential - Draft

B6.3 Types of cache

There are many different possible types of cache, that can be distinguished by implementation choices such as:

- cache size and associativity
- how they handle instruction fetches
- how they handle data writes.

Several of these implementation choices are detailed in the following subsections.. The System Control Coprocessor can contain a Cache Type Register, see section B_TBD_CP15_register0cache_type. The Cache Type Register is provided in ARMv6 and future implementations.

B6.3.1 Unified or separate caches

A memory system can use the same cache when processing instruction fetches as it does when processing data loads and stores. Such a cache is known as a *unified cache*. A unified cache and memory model is often referred to as a *von Neumann* architecture.

Alternatively, a memory system can use a different cache to process instruction fetches from the cache it uses to process data loads and stores. In this case, the two caches are known collectively as *separate caches* and individually as the *instruction cache* and *data cache*. The use of separate instruction and data caches, even with a unified main memory, is often referred to as a *Harvard* architecture.

The use of separate caches has the advantage that the memory system can often process both an instruction fetch and a data load/store in the same clock cycle, without a need for the cache memory to be multi-ported. The main disadvantage is that care must be taken to avoid problems caused by the instruction cache becoming out-of-date with respect to the data cache and/or main memory (see *Memory Coherency and Access Issues* on page B2-20).

It is also possible for a memory system to have an instruction cache but no data cache, or a data cache but no instruction cache. For the purpose of the memory system architectures, such a system is treated as having separate caches, where one cache is not present or has zero size.

B6.3.2 Write-through or write-back caches

When a cache hit occurs for a data store access, the cache line containing the data is updated to contain its new value. Because this cache line will eventually be re-allocated to another address, the new value must also be written to the main memory location for the data. There are two common techniques for handling this:

- In a *write-through* cache, the new data is also immediately written to the main memory location. This is usually done though a write buffer, to avoid slowing down the processor.

- In a *write-back* cache, the cache line is marked as *dirty*. This means that it contains data values that are more up-to-date than those in main memory. Whenever a dirty cache line is selected to be re-allocated to another address, the data currently in the cache line is written back to main memory. Writing back the contents of the cache line in this manner is known as *cleaning* the cache line, or a *victim write*. Another common term for a write-back cache is a *copy-back cache*.

Write-through caches can cause the processor to stall if it can generate data stores faster than they can be processed by the write buffer. The result is reduced system performance.

Write-back caches only store to main memory when a cache line is re-allocated, even if many stores have occurred to the cache line. Because of this, write-back caches normally generate fewer stores to main memory than write-through caches. This reduces the cache to main memory bandwidth requirement, and helps to alleviate the problem described above for write-through caches. However, write-back caches have a number of drawbacks, including:

- longer-lasting discrepancies between cache and main memory contents (see *Memory Coherency and Access Issues* on page B2-20)

- a longer worst-case sequence of main memory operations before a data load can be completed, that can increase the system's worst-case interrupt latency

- Cache cleaning might be necessary for correctness reasons as part of a cache and memory management policy.

- increased complexity of implementation.

Some write-back caches allow a choice to be made between write-back and write-through behavior (see *Cacheability and bufferability* on page B2-20).

B6.3.3 Read-allocate or write-allocate caches

There are two common techniques to deal with a cache miss on a data store access:

- In a *read-allocate* cache, the data is simply stored to main memory. Cache lines are only allocated to memory locations when data is read/loaded, not when it is written/stored.

- In a *write-allocate* cache, a cache line is allocated to the data and the current contents of main memory are read into it, then the data is written to the cache line. (It can also be written to main memory, depending on whether the cache is write-through or write-back.)

The main advantages and disadvantages of these techniques are performance-related. Compared with a read-allocate cache, a write-allocate cache can generate extra main memory read accesses that would not have otherwise occurred and/or save main memory accesses on subsequent stores because the data is now in the cache. The balance between these depends mainly on the number and type of the load/store accesses to the data concerned, and on whether the cache is write-through or write-back.

Prior to ARMv6, write-allocate or read-allocate caches used in an ARM memory system are IMPLEMENTATION DEFINED.

VMSAv6 defines the cache allocation policy as described in section B_TBD_VMSAv6_TEXbits.

B6.3.4 Replacement strategies

If a cache is not direct-mapped, a cache miss for a memory address requires one of the cache lines in the cache set associated with the address to be re-allocated. The way this cache line is chosen is known as the *replacement strategy* of the cache.

Two typical replacement strategies are:

Random replacement

> The cache control logic contains a pseudo-random number generator, the output of which is used to select the cache line to be re-allocated.

Round-robin replacement

> The cache control logic contains a counter that is used to select the cache line to be re-allocated. Each time this is done, the counter is incremented, so that a different choice is made next time.

There is a control bit in the System Control Coprocessor to allow ARM implementations to select one of two replacement choices, see section B_TBD_CWB_control_register. Typically, one choice is a simple, easily predictable strategy like round-robin replacement, with a random replacement algorithm as the alternative.

Some caches allow a choice of the replacement strategy in use. Typically, one choice is a simple, easily predictable strategy like round-robin replacement, that allows the worst-case cache performance for a code sequence to be determined reasonably easily. The main drawback of such strategies is that their average performance can change abruptly when comparatively minor details of the program change.

Round robin replacement strategies are more deterministic, but the performance can vary greatly with the data set. For example, suppose a program is accessing data items D1, D2, ..., Dn cyclically and that all of these data items happen to use the same cache set. With round-robin replacement in an m-way set-associative cache, the program is liable to get:

- nearly 100% cache hits on these data items when $n \leq m$
- 0% cache hits as soon as n becomes m+1 or greater.

In other words, a minor increase in the amount of data being processed can lead to a major change in how effective the cache is.

Random replacement has less easily predictable behavior. This makes the worst-case behavior harder to determine, but also makes the average performance of the cache vary more smoothly with parameters like working set size.

B6.4 L1 cache

The L1 Cache is the closest level of Cache to the CPU. It is the only level of Cache that is fully specified in ARMv6.

The L1 Cache can be implemented in a Harvard arrangement, with separate Instruction and Data Caches, or in a von Neumann arrangement, where all cached items, both instruction and data, are held in a unified structure. In a Harvard arrangement, an implementation does not need to include hardware support for coherency between the Instruction and Data caches. Where such support would be required, for example, in the case of self-modifying code, the software must make use of the cache cleaning instructions to avoid any such problems.

The L1 Cache must appear to software to behave as follows:

- the entries in the cache do not need to be cleaned and/or invalidated by software for different virtual to physical mappings

- aliases to the same physical address may exist in memory regions which are described in the page tables as being cacheable, subject to the restrictions for 4KB small pages outlined in *Restrictions on Page Table Mappings*.

Caches can be implemented with virtual or physical addressing, including indexing, provided these behaviors are met.

B6.4.1 Restrictions on Page Table Mappings

To aid implementations using virtually indexed, physically addressed caches and their handling of aliases, a restriction on the mapping of pages that remap virtual address bits [13:12] can be required. In this case, the need for the restriction is signified by setting bit 9 of the Cache Size field for the Instruction and Data caches in the Cache Type Register, see section B_TBD_CP15_reg0_cachetype.

This restriction allows these bits of the virtual address to be used to index into the cache without requiring hardware support to avoid alias problems.

For pages marked as non-shared, if bit 9 of the Cache Size field is set, that is, the restriction applies to pages which remap virtual address bits[13:12], then to prevent aliasing problems when 4KB pages are used one of the two following sets of restrictions shall apply:

- If multiple virtual addresses are mapped onto the same physical addresses, for all mappings bits [13:12] the virtual address must be equal, and equal bits [13:12] of the physical address. The same physical address can be mapped by TLB entries of different page sizes, including page sizes over 4KB.

- If all mappings to a physical address are of a page size equal to 4KB, then the restriction that bits [13:12] of the virtual address must equal bits [13:12] of the physical address is not necessary. Bits [13:12] of all virtual address aliases must still be equal.

There is no restriction on the more significant bits in the virtual address equalling those in the physical address.

———— **Note** ————

In ARMv6 1KB (Tiny) pages are OBSOLETE. See section *Key changes introduced in VMSAv6* on page B4-3.

B6.4.2 Considerations for additional levels of cache

Additional levels of cache can be implemented in a system. Guidelines for the form and control of an L2 cache are provided in sections B_TBD_memhierarchy_L2, B_TBD_CP15_L2cache and B_TBD_CWB_L2cache. Other cache level architectures are not mandated within this architecture, but they must appear to software to be unified to the memory system. This is to permit the writing of self-modifying code without the need for cache cleaning and invalidation routines, beyond those described in section B_TBD_CWB_clean_routine.

It is recommended that all levels of cache have cache control functions that provide at a minimum the following features:

• Cache Cleaning Routines

• Cache Invalidation Routines

Section B_TBD_MMU_TEXdescription describes the Inner and Outer attributes in the MMU. These attributes are defined for each page. They are used to control the caching policy at different cache levels for different regions of memory. The Inner attributes are used to define the caching policy at Level 1. Implementations may use the Inner and Outer attributes to describe caching policy at other levels in an IMPLEMENTATION DEFINED manner.

B6.5   CP15 registers

In ARMv6, the type of cache can be determined from the System Coprocessor register 0, and controlled through registers 1, 7 and 9. In earlier architecture variants, it is IMPLEMENTATION DEFINED which of these features are present. All registers are privileged access only unless otherwise defined. Unimplemented registers are UNDEFINED.

──── **Note** ────

Prior to ARMv6, a System Coprocessor was not mandatory. Earlier architecture variants might have implemented a subset of the functionality or used it in an IMPLEMENTATION DEFINED manner. However, previous versions of the architecture have documented System Coprocessor register (CP15) features. Where new features have been added, or definitions of pre-existing features changed, this is explicitly stated.

B6.5.1   Register 0: cache type

The Cache size and organization is implementation specific. This read-only register describes the organization and size of the cache. This provides information to an operating systems about how to perform operations such as cache cleaning and lockdown. Reading CP15 register 0 with the opcode_2 field set to 1 accesses the Cache Type register.

```
MRC p15, 0, Rd, c0, c0, 1 ; returns Cache Type register
```

The Cache Type register supplies the following details about the cache:
*   whether it is a unified cache or separate caches
*   size, line length, and associativity
*   page mapping requirements (ARMv6 only)
*   whether it is a write-through cache or a write-back cache
*   how it can be cleaned efficiently (in the case of a write-back cache)
*   whether cache lock-down is supported.

B6.5.2   Cache Type register

The Cache Type register supplies the following details about the cache:

*   whether it is a unified cache or separate instruction and data caches
*   its size, line length and associativity
*   whether it is a write-through cache or a write-back cache
*   cache cleaning and lockdown capabilities.

The format of the Cache Type register is:

31 29	28 25	24 23	23 12	11 0
0 0 0	ctype	S	Dsize	Isize
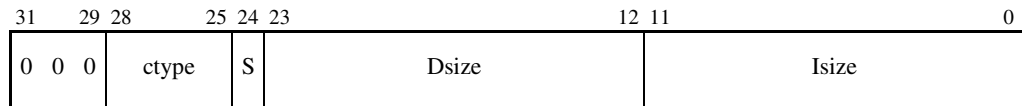
ctype   Specifies details of the cache not specified by the S bit and the Dsize and Isize fields. See Table B6-1 for details of the encoding. All values not specified in the table are reserved for future expansion.

S bit   Specifies whether the cache is a unified cache (S == 0), or separate instruction and data caches (S == 1). If S == 0, the Isize and Dsize fields both describe the unified cache, and must be identical.

Dsize   Specifies the size, line length and associativity of the data cache, or of the unified cache if S == 0. See *Cache size fields* on page B6-14 for details of the encoding.

Isize   Specifies the size, line length and associativity of the instruction cache, or of the unified cache if S == 0. See *Cache size fields* on page B6-14 for details of the encoding.

Table B6-1 Cache type values

ctype field	Method	Cache cleaning	Cache lock-down
0b0000	Write-through	Not needed	Not supported
0b0001	Write-back	Read data block	Not supported (deprecated in ARMv6)
0b0010	Write-back	Register 7 operations	Not supported (deprecated in ARMv6)
0b0110	Write-back	Register 7 operations	Format A
0b0111	Write-back	Register 7 operations	Format B (deprecated in ARMv6)
0b1110	Write-back	Register 7 operations	Format C
0b0101	Write-back	Register 7 operations	Format D

Copyright © 1996–2003 ARM Limited. All rights reserved.
Confidential - Draft

The *Read data block* method of cleaning write-back caches encoded by ctype == 0b0001 consists of loading a sequential block of data with size equal to that of the cache, and which is known not to be in the cache already. It is only suitable for use when the cache organization guarantees that this causes the entire cache to be reloaded. (For example, direct-mapped caches normally have this property, as do caches using some types of round-robin replacement.)

——— **Note** ———

This method of cache cleaning must *only* be used if the Cache Type register has ctype == 0b0001, or if implementation documentation states that it is a valid method for the implementation.

Register 7: cache management functions on page B6-18 gives details of the register 7 operations used for cleaning other write-back caches.

For an explanation of cache lockdown and of the formats referred to in Table B6-1 on page B6-13, see *Register 9: cache lockdown functions* on page B6-27.

B6.5.3 Cache size fields

The Dsize and Isize fields in the Cache Type register have the same format, as follows:

	11	9	8		6	5		3	2	1	0
	0	0	P	size		assoc			M	len	
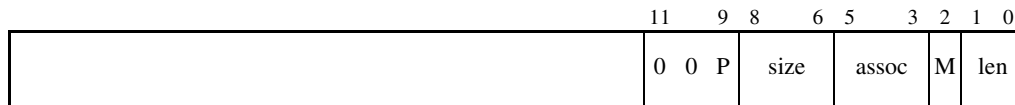
Bits[11:10] are reserved for future expansion.

Bit[9] (P-bit) indicates whether a restriction exists on page allocation concerning bits[13:12] of the virtual address:

0   no restriction

1   restriction applies, see *Restrictions on Page Table Mappings* on page B6-10.

The size of the cache is determined by the size field and M bit, as shown in Table B6-2.

Table B6-2 Cache sizes

size field	Size if M == 0	Size if M == 1
0b000	0.5KB	0.75KB
0b001	1KB	1.5KB
0b010	2KB	3KB
0b011	4KB	6KB
0b100	8KB	12KB

Table B6-2 Cache sizes

size field	Size if M == 0	Size if M == 1
0b101	16KB	24KB
0b110	32KB	48KB
0b111	64KB	96KB

The line length of the cache is determined by the len field, as shown in Table B6-3.

Table B6-3 Cache line lengths

len field	Cache line length
0b00	2 words (8 bytes)
0b01	4 words (16 bytes)
0b10	8 words (32 bytes)
0b11	16 words (64 bytes)

The associativity of the cache is determined by the assoc field and the associativity modifier (M-bit), as shown in Table B6-4.

Table B6-4 Cache associativity

assoc field	Associativity if M == 0	Associativity if M == 1
0b000	1-way (direct-mapped)	cache absent
0b001	2-way	3-way
0b010	4-way	6-way
0b011	8-way	12-way
0b100	16-way	24-way
0b101	32-way	48-way
0b110	64-way	96-way
0b111	128-way	192-way

The *cache absent* encoding overrides all other data in the cache size field.

Alternatively, the following formulae can be used to determine the values LINELEN, ASSOCIATIVITY and NSETS, defined in *Cache size* on page B6-6, once the *cache absent* case (assoc == 0b000, M == 1) has been checked for and eliminated:

```
LINELEN        = 1 << (len+3)              /* In bytes */

MULTIPLIER     = 2 + M
ASSOCIATIVITY  = MULTIPLIER << (assoc-1)

NSETS          = 1 << (size + 6 - assoc - len)
```

Multiplying these together gives the overall cache size as:

```
CACHE_SIZE     = MULTIPLIER << (size+8)    /* In bytes */
```

——— **Note** ———

Cache length fields with (size + 6 - assoc - len) < 0 are invalid, as they correspond to impossible combinations of line length, associativity and overall cache size. So the formula for NSETS never involves a negative shift amount.

B6.5.4   Register 1: cache and write buffer control bits

The following bits in register 1 of the System Control Coprocessor control caches and write buffers:

C (bit[2])   If a unified cache is used, this is the enable/disable bit for the unified cache. If separate
caches are used, this is the enable/disable bit for the data cache. In either case:

0 = Cache disabled

1 = Cache enabled.

If the cache is not implemented, this bit *reads as 0* (RAZ) and ignores writes. If the cache
is implemented, it must be possible to disable it by setting this bit to 0.

The C bit must reset to 0. Behavior prior to ARMv6 might differ.

W (bit[3])   This is the enable/disable bit for the write buffer:

0 = Write buffer disabled

1 = Write buffer enabled.

If the write buffer is not implemented, this bit RAZ and ignore writes. If the write buffer is
implemented, it must be possible to disable it by setting this bit to 0.

The W bit must reset to 0. Behavior prior to ARMv6 might differ.

I (bit[12])   If separate caches are used, this is the enable/disable bit for the instruction cache:

0 = Cache disabled

1 = Cache enabled.

If a unified cache is used or the instruction cache is not implemented, this bit RAZ and
ignores writes. If the instruction cache is implemented, it must be possible to disable it by
setting this bit to 0.

The I bit must reset to 0. Behavior prior to ARMv6 might differ.

RR (bit[14])   If the cache allows the use of an alternative replacement strategy that has a more easily
predictable worst-case performance, this bit selects it:

0 = Normal replacement strategy (for example, random replacement)

1 = Predictable strategy (for example, round-robin replacement).

The RR bit must reset to 0. Behavior prior to ARMv6 might differ.

For a full description of the System Control Coprocessor register 1, see section B_TBD_CP15_register1.

B6.5.5 Register 7: cache management functions

The System Control coprocessor register 7 is a write-only register that is used to control L1 caches and write buffers. It is also used to implement some similar functions on prefetch buffers and branch target caches, if they exist, and to implement the *wait for interrupt* clock control function. Table B6-6 on page B6-20 lists the functions available.

All functions are invoked using the following instruction format:

```
MCR p15,0,<Rd>,c7,<CRm>,<opcode2>
```

Writing to register 7 with a combination of `<CRm>` and `<opcode2>` that is not listed in Table B6-6 on page B6-20 has UNPREDICTABLE results.

Most CP15 Register 7 operations can only be executed in a privileged mode. A small number of instructions can also be executed in User mode, marked [c] in Table B6-6 on page B6-20. Attempting to execute a privileged operation in user mode will result in an UNDEFINED instruction exception.

All of the functions in Table B6-6 on page B6-20 for a given cache organization must be implemented by an implementation that uses that organization. Other functions must have no effect.

In Table B6-6 on page B6-20, the following terms apply:

Clean Applies to write-back data caches, and means that if the cache line contains stored data that has not yet been written out to main memory, it is written to main memory now, and the line is marked as clean.

Invalidate Means that the cache line (or all the lines in the cache) is marked as invalid. No cache hits can occur for that line until it is re-allocated to an address.

For write-back data caches, this does not include cleaning the cache line unless that is also stated.

Prefetch Means the memory cache line at the specified virtual address is loaded into the cache. In ARMv6 there is no alignment requirement for the virtual address. Prior to ARMv6 the address was required to be cache line aligned.

Data write barrier DWB. See section B_TBD_memorder_DWB for the new ARMv6 definition. This instruction completes when all *explicit* write transactions occurring in program order before this instruction are completed. No instructions occurring in program order after this instruction are executed until this instruction completes.

Data write barrier can be executed in both privileged and user modes of operation.

Explicit memory accesses are reads or writes to the memory system that are generated by load or store instructions being executed in the CPU. The following are not explicit accesses:
- reads or writes generated by L1 DMA transactions
- instruction fetches
- accesses caused by a hardware page table access.

Data memory barrier

DMB. Introduced in ARMv6, and described in section B_TBD_memorder_DMB. This instruction completes when all explicit memory transactions occurring in program order before this instruction are completed. No instructions occurring in program order after this instruction are executed until this instruction completes.

DMB can be executed in both privileged and user modes.

Wait for interrupt Puts the ARM into a low power state and stops it executing further until an interrupt, or a debug request, occurs. An interrupt restarts execution even if disabled by the mask in the CPSR.

When an interrupt does occur, the MCR instruction completes and the IRQ or FIQ handler is entered as normal. The return link in R14_irq or R14_fiq contains the address of the MCR instruction plus 8, so that the normal instruction used for interrupt return (SUBS PC,R14,#4) returns to the instruction following the MCR.

Flush prefetch buffer

Flushing the instruction prefetch buffer has the effect that all instructions occurring in program order after this instruction are fetched from the memory system, including the L1 cache or TCM, after the execution of this instruction. This operation can be useful for ensuring the correct execution of self-modifying code.

Flush Prefetch Buffer can be executed in both privileged and user modes.

Data Is the value that is written to register 7. This is the value in the register <Rd> specified in the MCR instruction.

From ARMv6, if the data is stated to be a virtual address, it does not need to be cache line aligned. The address is looked up in the cache for the particular operations. Invalidation and cleaning operations have no effect if they miss in the cache. If the corresponding entry is not in the TLB, these instructions might cause a hardware page table walk.

For the cache control operations, the virtual addresses that are passed to the cache are not translated by the FCSE extension. A *Modified Virtual Address* (MVA) must be used as described in section B_TBD_FCSE_MVA.

A loop of single line cache control operations can be used to clean and/or invalidate all cache lines relating to a specified range of addresses.

If the data is stated to be set/way, the data identifies the cache line that the operation is to be applied to by specifying which cache set it belongs to and the way number within the set. These values are defined in section B_TBD_CWB_set_assoc.

A loop of operations of this type can be used to clean and/or invalidate all of the cache.

The format of set/way data is shown in Table B6-5 on page B6-20, where L, A, and S are the logarithms base 2 of the cache size parameters LINELEN, ASSOCIATIVITY and NSETS, rounded up to an integer in the case of A. These parameters can be found in the Cache Type register. NSETS is derived from the size

information using the other two parameters. The TC field in the Cache Type register indicates whether the Data should apply to the cache or to any TCM configured as SmartCache, described in section B_TBD_TCM_smartcache.

Table B6-5 Set/Way data register values

31	32-A	31-A	L+S	L+S-1	L	L-1	0

index	SBZ/UNP	set	SBZ	TC

Table B6-6 Register 7: cache control and similar functions

<CRm>	<opcode2>	Function	Data	
c0	4	Wait for interrupt	SBZ	
c5	0	Invalidate entire instruction cache (flush branch target cache, if applicable)	SBZ	a
c5	1	Invalidate instruction cache line	MVA[b]	a
c5	2	Invalidate instruction cache line	Set/way	a
c5	4	Flush prefetch buffer (PrefetchFlush)	SBZ	c
c5	6	Flush entire branch target cache (if applicable)	SBZ	
c5	7	Flush branch target cache entry (if applicable)	MVA[b]	
c6	0	Invalidate entire data cache	SBZ	d
c6	1	Invalidate data cache line	MVA[b]	d
c6	2	Invalidate data cache line	Set/way	d
c7	0	Invalidate both instruction and data caches or unified cache (flush branch target cache, if applicable)	SBZ	e
c7	1	Invalidate unified cache line	MVA[b]	f
c7	2	Invalidate unified cache line	Set/way	f
c10	0	Clean entire data cache	SBZ	d
c10	1	Clean data cache line	MVA[b]	d
c10	2	Clean data cache line	Set/way	d
c10	4	Data Write Barrier (formerly Drain Write Buffer)	SBZ	c

Table B6-6 Register 7: cache control and similar functions (continued)

<CRm>	<opcode2>	Function	Data	
c10	5	Data Memory Barrier (Introduced with ARMv6. May be applied to earlier architecture variants.)		c
c11	0	Clean entire unified cache	SBZ	f
c11	1	Clean unified cache line	MVA[b]	f
c11	2	Clean unified cache line	Set/way	f
c13	1	Prefetch instruction cache line	MVA[b]	
c14	0	Clean and Invalidate entire data cache	SBZ	d
c14	1	Clean and invalidate data cache line	MVA[b]	d
c14	2	Clean and invalidate data cache line	Set/way	d
c15	0	Clean and invalidate entire unified cache	SBZ	f
c15	1	Clean and invalidate unified cache line	MVA[b]	f
c15	2	Clean and invalidate unified cache line	Set/way	f

a. Only applies to a separate instruction cache.
b. *Modified Virtual Address* (MVA) is described in section B_TBD_FCSE_MVA.
c. Available in User mode.
d. Only applies to a separate data cache.
e. Applies to unified and separate caches.
f. Only applies to a unified cache.

The cache invalidation operations apply to all cache locations, including those locked in the cache.

An explicit flush of the entire Branch Target Cache must be performed after the invalidation of instruction cache lines, otherwise the result is UNPREDICTABLE.

Clean and Clean & Invalidate Entire Data (or Unified) Cache operations

The CP15 register 7 specifies operations for cleaning the entire data (or unified) cache, and also for performing a clean and invalidate of the entire data (or unified) cache. These are *blocking operations* (see *Block transfer operations using CP15 Register 7* on page B6-23) that can be interrupted. If they are interrupted, the R14 value that is captured on the interrupt is the address of the instruction that launched the cache clean operation + 4. This allows the standard return mechanism for interrupts to restart the operation.

If it is essential that the cache is clean (or clean and invalid) for a particular operation, the sequence of instructions for cleaning (or cleaning and invalidating) the cache for that operation must allow for the arrival of an interrupt at any time that interrupts are not disabled. This is because interrupts might write to a previously cleaned cache block. For this reason, the Cache Dirty Status register indicates whether the cache has been written to since the last clean of the cache was started.

The Cache Dirty Status register is a read only register. To access it, use the following instruction:

```
MRC p15, 0, Rd, c7, c10, 6
```

The format of the Cache Dirty Status register is shown in Table B6-7.

Table B6-7 Cache Dirty Status Register

31	1	0
SBZ/UNP		C

C (bit[0])   Cache Dirty Status

   0   No write has hit the cache since the last cache clean or reset successfully left the cache clean

   1   The cache may contain dirty data.

The Cache Dirty Status register can be interrogated to determine whether the cache is clean, and if this is done while interrupts are disabled, subsequent operation(s) can rely on having a clean cache. The following sequence illustrates this approach.

```
       ; interrupts are assumed to be enabled at this point

Loop1   MOV R1, #0
        MCR CP15, 0, R1, C7, C10, 0  ; Clean (or Clean & Invalidate) Cache
        MRS R2, CPSR
        CPSID iaf                    ; Disable interrupts
        MRC CP15, 0, R1, C7, C10, 6  ; Read Cache Dirty Status Register
        ANDS R1, R1, #01             ; Check if it is clean
        BEQ UseClean
        MSR CPSR, R2                 ; Re-enable interrupts
        B Loop1                      ; - clean the cache again
UseClean    Do_Clean_Operations      ; Perform whatever operation relies on the cache
                                     ; being clean/invalid. To reduce impact on
                                     ; interrupt latency, this sequence should be short
        MSR CPSR, R2                 ; Re-enable interrupts
```

——— **Note** ———

The long Cache Clean operation is performed with interrupts enabled throughout this routine.

ARM DDI 0100F-02

—— **Note** ——

Implementation: because the Cache Dirty Status register is used to indicate whether the cache is still clean, it is not necessary for an implementation to restart the cache clean (or clean and invalidate) operations from the first line of the cache. Implementations must ensure that, if the clean and invalidate operation does not restart from the start of the cache, no data is lost by the invalidation step.

B6.5.6 Block transfer operations using CP15 Register 7

The block operations shown in Table B6-8 can optionally be supported using CP15 register 7. Block operations were introduced into the architecture with ARMv6. If the operations are not implemented, then they must cause an Undefined instruction exception. Implementations that support SmartCache (see section B_TBD_TCM_smartcache) behavior must implement all these instructions.

Table B6-8 Block transfer operations

Operation	Blocking a or non-blocking	Instruction or Data	User or Privileged	Exception Behavior
Prefetch Range	Non-Blocking	Instruction or Data	User/Privileged	None
Clean Range	Blocking	Data only	User/Privileged	Data Abort
Clean and Invalidate Range	Blocking	Data only	Privileged	Data Abort
Invalidate Range	Blocking	Instruction or Data	Privileged	Data Abort

a. The cache block transfer operations for cleaning and/or invalidating a range of addresses from the cache are blocking operations. Following instructions must not be executed until this operation has completed. A non-blocking operation can permit following instructions to be executed before the operation is completed. In the event of an exception occurring a non-blocking operation does not signal an exception to the core. This allows implementations to retire following instructions while the non-blocking operation is executing, without the need to retain precise processor state.

Each of the range operations is started using an MCRR instruction. The data of the two registers is used to specify the Block Start Address and the Block End address. All block operations are performed on the cache (or SmartCache) lines that include the range of addresses between the Block Start Address and Block End Address inclusive. If the Block Start Address is greater than the Block End Address the effect is UNPREDICTABLE.

Only one block transfer at a time is supported. Attempting to start a second block transfer while a first block transfer is in progress causes the first block transfer to be abandoned and the second block transfer to be started. The Block Transfer Status register indicates whether a block transfer is in progress. This can be used to prevent waiting if it is not desired. It is expected that block transfers are stopped on a context switch.

All block transfers are interruptible. When blocking transfers are interrupted, the R14 value that is captured is the address of the instruction that launched the block operation + 4. This allows the standard return mechanism for interrupts to restart the operation.

For performance reasons, it is expected that implementations allow following instructions to be executed while a non-blocking Prefetch Range instruction is being executed. In such implementations, the R14 value captured on an interrupt is determined by the execution state presented to the interrupt in following instruction stream. However, implementations that treat a Prefetch Range instruction as a blocking operation must capture the R14 value as described in the previous paragraph.

If the FCSE PID (section B_TBD_FCSE_CP15reg) is changed while a prefetch range operation is running, it is UNPREDICTABLE at which point this change is seen by the prefetch range.

Exception behavior

The blocking block transfers cause a data abort on a translation fault if a valid page table entry cannot be fetched. The CP15 FAR indicates the address that caused the fault, and the CP15 FSR indicates the reason for the fault.

Any fault on a prefetch range operation results in the operation failing without signalling an error.

Register encodings

Block operations are supported using CP15 register 7 instructions as shown in Table B_TBD_CWB_blockops. These operations can only be performed using an MCRR instruction. All other operations to these registers are ignored.

The instruction format is as follows:

```
MCRR p15, Opcode, Rd, Rn, CRm
```

Table B6-9 Enhanced Cache Control Operations using MCRR

<CRm>	Opcode	Function	Rn Data (VA [a])	Rd Data (VA [a])
c5	0	Invalidate Instruction Cache Range [b]	Start Address	End Address
c6	0	Invalidate Data Cache Range [b]	Start Address	End Address
c12	0	Clean Data Cache Range [c]	Start Address	End Address
c12	1	Prefetch Instruction Range [c]	Start Address	End Address
c12	2	Prefetch Data Range [c]	Start Address	End Address
c14	0	Clean and Invalidate Data Cache Range [b]	Start Address	End Address

a. The true virtual address, before any modification by the Fast Context Switch Extension (see chapter B_TBD_FCSE). This address is translated by the FCSE logic.
b. Accessible only in privileged modes. Result in an UNDEFINED instruction exception if the operation is attempted in user mode.
c. Accessible in both user and privileged modes.

Each of the Range operations operates between cache (or SmartCache) lines containing the Start Address and the End Address, inclusive of start address and end address.

The Start Address and End Address data values passed by the MCRR instructions have the format shown in Table B6-10, where L is the logarithm base 2 of the cache size parameter LINELEN. Because the least significant address bits are ignored, the transfer automatically adjusts to a line length multiple spanning the programmed addresses.

Table B6-10 Block Address register

31	L	L–1	0
Virtual address		IGN	

Start address   **Virtual Address (Bits[31:L])**
the first virtual address of the block transfer

End address   **Virtual Address (Bits[31:L])**
the virtual address at which the block transfer stops (this address is at the start of the line containing the last address to be handled by the block transfer).

——— **Note** ———

Only these block operations use true virtual addresses. All other address-based cache operations use MVAs.

Two additional CP15 register 7 operations are provided to support block transfer management:

StopPrefetchRange   `MCR p15, 0, Rd, c7, c12, 5 ; Rd SBZ`

PrefetchStatus   `MRC p15, 0, Rd, c7, c12, 4 ; Rd returns the status`

Both operations are accessible in User and privileged modes. Because all block operations are mutually exclusive, that is, only one operation can be active at any time, the PrefetchStatus operation returns the status of the last issued Prefetch request, instruction or data.

The Block Transfer Status register has the format shown in Figure B_TBD_CWB_BTSreg.

Table B6-11 Block Transfer Status register

31	1	0
SBZ/UNP		R

R (bit[0])   Block Prefetch Running

0   no prefetch in operation

1   prefetch in operation

B6.5.7 Cache cleaning and invalidating operations for TCM configured as SmartCache

All cache line and block cleaning and invalidation operations based on virtual address, as defined in CP15 register 7, include TCM regions that are configured as SmartCache, see section B_TBD_TCM_smartcache.

The Set/Way operations are supported for the TCMs operating as SmartCache. In this case, the way number is taken to be the TCM number, and the meaning of the set number is unchanged. To distinguish between these operations as applied to the Cache and as applied to TCM, the bottom bit of the set/way data register is used, as shown in Figure B_TBD_setway.

The line length of the TCM operating as SmartCache must be the same as the Cache Line length, defined in the Cache Type register:

TC (Bit[0]) TCM bit. Indicates that this register is referring to the TCMs rather than the Cache.

    0   Register refers to cache.

    1   Register refers to TCM

Invalidate, clean, and clean+invalidate entire cache operations have no effect on TCMs operating as SmartCache.

B6.5.8   Register 9: cache lockdown functions

One problem with caches is that although they normally improve average access time to data and instructions, they usually increase the worst-case access time. This occurs for a number of reasons, including:

*   There is a delay before the system determines that a cache miss has occurred and starts the main memory access.
*   If a write-back cache is being used, there might be a further delay because of the need to store the contents of the cache line that is being re-allocated.
*   A whole cache line is loaded from main memory, not just the data requested by the ARM processor.

In real-time applications, this increase in the worst-case access time can be very significant.

Cache lockdown is a feature of most ARM memory systems designed to alleviate this. It allows critical code and data (for example, high-priority interrupt routines and the data they access) to be loaded into the cache in such a way that the cache lines containing them are not subsequently re-allocated. This ensures that all subsequent accesses to the code and data concerned are cache hits and therefore complete quickly.

The ARM architecture supports four formats for the cache lockdown mechanism, known as Format A, Format B, Format C and Format D. The Cache Type register in the System Control coprocessor (CP15 register 0) contains information on the lockdown mechanism adopted, see section B_TBD_CP15_cache_type_reg.

——— **Note** ———

Format B is deprecated in ARMv6.

Formats A, B, and C all operate on cache ways (see section B_TBD_CWB_cache_set_assoc). Format D is a cache entry locking mechanism.

General conditions applying to Format A, B & C lockdown

The instructions used to access the CP15 register 9 lockdown registers are as follows:

```
MCR p15, 0, Rd, c9, c0, 0    ; write unified/data lockdown register
MRC p15, 0, Rd, c9, c0, 0    ; read unified/data lockdown register
MCR p15, 0, Rd, c9, c0, 1    ; write instruction lockdown register
MRC p15, 0, Rd, c9, c0, 1    ; read instruction lockdown register
```

LINELEN, ASSOCIATIVITY and NSETS are the cache size parameters described in *Cache size* on page B6-6. A cache way consists of one cache line from each cache set and is labelled from 0 to ASSOCIATIVITY-1. Formats A, B, and C all use cache ways for lockdown granularity (the lockdown block). A cache locking scheme can use any number of lockdown blocks from 1 to ASSOCIATIVITY-1. If N lockdown blocks are locked down, they have indices 0 to N-1, and lockdown blocks N to ASSOCIATIVITY-1 are available for normal cache operation.

A cache way based lockdown implementation must not lock down the entire cache. At least one lockdown block must be left for normal cache operation. Failure to adhere to this restriction results in UNPREDICTABLE behavior.

Each lockdown block can hold NSETS memory cache lines, provided each of the memory cache lines is associated with a different cache set. Although caches operate on aliased address spaces, it is recommended that lockdown systems are designed such that each lockdown block is considered as a set of NSETS consecutive memory cache lines. This is NSETS × LINELEN consecutive memory locations, starting at a cache line boundary. (Such sets are easily identified and are guaranteed to consist of one cache line associated with each cache set.)

For lockdown purposes, the cache is divided into *lockdown blocks*, each of which consists of one line from each cache set. The lockdown blocks are indexed from 0 to ASSOCIATIVITY-1. The cache lines in a lockdown block are chosen to have the same WAY number as the lockdown block (see *Set-associativity* on page B6-5). So lockdown block *n* consists of the cache line with index *n* from each cache set, for *n* from 0 to ASSOCIATIVITY-1.

Cache lockdown can be performed with a granularity of a lockdown block. Any number of lockdown blocks from 1 to ASSOCIATIVITY-1 can be chosen. If N lockdown blocks are locked down, they have indices 0 to N-1, and lockdown blocks N to ASSOCIATIVITY-1 are available for normal cache operation.

Each lockdown block can hold NSETS memory cache lines, provided each of the memory cache lines is associated with a different cache set. It is recommended that systems are designed so that each lockdown block contains a set of NSETS consecutive memory cache lines. This is NSETS × LINELEN consecutive memory locations, starting at a cache line boundary. (Such sets are easily identified and are guaranteed to consist of one cache line associated with each cache set.)

Formats A and B lockdown

Formats A and B use a WAY field that is chosen to be wide enough to hold the way number of any lockdown block. Its width W is the logarithm base 2 of ASSOCIATIVITY, rounded up to the nearest integer if necessary.

A Format A Lockdown register has the form shown in Table B6-12.

Table B6-12 Format A lockdown register

31 32–W	31–W 0
WAY	SBZ/UNP

Reading a Format A register returns the value last written to it.

Writing a Format A register has the following effects:

- The next cache miss in each cache set replaces the cache line with the specified WAY in that cache set.

- The replacement strategy for the cache is constrained so that it can only select cache lines with the specified WAY and higher, until the register is written again.

A Format B Lockdown register has the form shown in Table B6-13.

Table B6-13 Format B lockdown register

31	30	W	W–1	0
L	SBZ/UNP		WAY	

Reading a Format B register returns the value last written to it.

Writing a Format B register has the following effects:

- If L == 1, all cache misses replace the cache line with the specified WAY in the relevant cache set until the register is written again.

- If L == 0, then:

    — If the previous value of L was 0 and the previous value of WAY is smaller than the new value, the behavior is UNPREDICTABLE.

    — Otherwise, the replacement strategy for the cache is constrained so that it can only select cache lines with the specified WAY and higher, until the register is written again.

Format A and B cache lockdown procedure

The procedure to lock down N lockdown blocks is as follows:

1. Ensure that no processor exceptions can occur during the execution of this procedure, for example by disabling interrupts. If for some reason this is not possible, all code and data used by any exception handlers that can get called must be treated as code and data used by this procedure for the purpose of steps 2 and 3.
2. If an instruction cache or a unified cache is being locked down, ensure that all the code executed by this procedure is in an uncacheable area of memory.
3. If a data cache or a unified cache is being locked down, ensure that all data used by the following code is in an uncacheable area of memory, apart from the data that is to be locked down.
4. Ensure that the data/instructions that are to be locked down are in a cacheable area of memory.
5. Ensure that the data/instructions that are to be locked down are not already in the cache, using cache clean and/or invalidate instructions as appropriate.
6. For each of i = 0 to N-1:
    a. Write to register 9 with WAY == i (for Formats A and B), and L == 1 (for Format B).
    b. For each of the cache lines to be locked down in lockdown block i:

        If a data cache or a unified cache is being locked down, use an LDR instruction to load a word from the memory cache line. This ensures that the memory cache line is loaded into the cache.

        If an instruction cache is being locked down, use the register 7 *prefetch instruction cache line* operation (<CRm> == c13, <opcode2> == 1) to fetch the memory cache line into the cache.
7. Write to register 9 with WAY == N (for Formats A and B), and L == 0 (for Format B).

——— **Note** ———

If the *Fast Context Switch Extension* (FCSE) described in TBD xref is being used, care must be taken in step 6b. This is because:

- If a data cache or a unified cache is being locked down, the address used for the LDR instruction is subject to modification by the FCSE.

- If an instruction cache is being locked down, the address used for the register 7 operation is being treated as data and so is not subject to modification by the FCSE.

To minimize the possible confusion caused by this, it is recommended that the lockdown procedure should:

- start by disabling the FCSE (by setting the PID to zero)

- where appropriate, generate modified virtual addresses itself by ORing the appropriate PID value into the top 7 bits of the virtual addresses it uses.

Format A and B cache unlock procedure

To unlock the locked-down portion of the cache, write to register 9 with WAY == 0 (for Formats A and B), and L == 0 (for Format B).

——— **Note** ———

Format B is deprecated in ARMv6.

Format C lockdown

Cache Lockdown Format C is a different form of cache way based locking. It allows the allocation to each cache way to be disabled or enabled. This provides some additional control over the cache pollution caused by particular applications, as well as a traditional lockdown function for locking critical regions into the cache.

A locking bit for each cache way determines whether the normal cache allocation mechanisms (either random or round-robin) are allowed to access that cache way.

For caches of higher associativity, only cache ways 0-31 may be locked.

A maximum of N-1 ways of an N-WAY cache can be locked. This ensures that a normal cache line replacement can be performed. If there are no cache ways which have L==0, this leads to UNPREDICTABLE behavior in handling a cache miss.

The 32 bits of the lockdown register (instruction or data dependent on the value of opcode2) determine the L bit for the associated cache way.

The cache lockdown register is normally modified in a read-modify-write sequence. For example, the following sequence sets the L bit to 1 for way 0 of the instruction cache:

```
MRC p15, 0, Rn, c9, c0, 1
ORR Rn, Rn, 0x01
MCR p15, 0, Rn, c9, c0, 1    ; set way 0 L-bit for the Icache
```

The format of the cache lockdown register is as shown in Figure B_TBD_CWB_formatCreg.

Table B6-14 Format C lockdown register

31   0

One L bit for each cache way
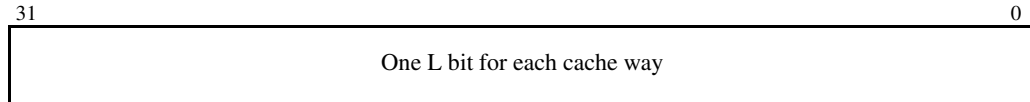
Bits[31:0]   The L bits for each cache way. If a cache way is not implemented, then the L bit for that way reads as 1, and writes to that bit are ignored. Each bit relates to its corresponding cache way, that is bit N refers to way N.

        0   Allocation to the cache way is determined by the standard replacement algorithm (reset state)

        1   No Allocation is performed to this cache way.

The Format C Lockdown Register should only be changed when it is certain that all outstanding accesses that could cause a cache line fill have completed. For this reason, a DrainWriteBuffer instruction should be executed before the Lockdown Register is changed.

Format C cache lock procedure

The procedure to lock down into a cache way i with N cache ways using Format C involves making it impossible to allocate to any cache way other than the target cache way *i*. This is the architecturally defined method for locking data into the caches:

1. Ensure that no processor exceptions can occur during the execution of this procedure, by disabling interrupts, for example. If for some reason this is not possible, all code and data used by any exception handlers that can get called must be treated as code and data used by this procedure for the purpose of steps 2 and 3.

2. If an instruction cache or a unified cache is being locked down, ensure that all the code executed by this procedure is in an uncacheable area of memory (including the Tightly-Coupled Memory) or in an already locked cache way.

3. If a data cache or a unified cache is being locked down, ensure that all data used by the following code (apart from the data that is to be locked down) is in an uncacheable area of memory (including the Tightly-Coupled Memory) or is in an already locked cache way.

4. Ensure that the data/instructions that are to be locked down are in a cacheable area of memory.

5. Ensure that the data/instructions that are to be locked down are not already in the cache, using cache clean and/or invalidate instructions as appropriate.

6. Write to register 9, <CRm> == 0, setting L==0 for bit *i* and L==1 for all other ways. This enables allocation to the target cache way.

7. For each of the cache lines to be locked down in cache way *i*:

   • If a data cache or a unified cache is being locked down, use an `LDR` instruction to load a word from the memory cache line, which ensures that the memory cache line is loaded into the cache.

   • If an instruction cache is being locked down, use the register 7 prefetch instruction cache line operation (<CRm> == c13, <opcode2> == 1) to fetch the memory cache line into the cache.

8. Write to register 9, <CRm> == 0 setting L == 1 for bit *i* and restoring all the other bits to the values they had before this routine was started.

Format C cache unlock procedure

To unlock the locked-down portion of the cache, write to register 9 setting L == 0 for each bit.

Format D lockdown

This format locks individual L1 cache line entries rather than using a cache way scheme. The methods differ for the instruction and data caches.

The instructions used to access the CP15 register 9 Format D lockdown registers are as follows:

```
MCR p15, 0, Rd, c9, c5, 0    ; fetch and lock instruction cache line,
                             ; Rd = MVA
MCR p15, 0, Rd, c9, c5, 1    ; unlock instruction cache,
                             ; Rd ignored
MCR p15, 0, Rd, c9, c6, 0    ; write data cache lock register,
                             ; Rd = set/clear lock mode
MRC p15, 0, Rd, c9, c6, 0    ; read data cache lock register,
                             ; Rd = lock mode status
MCR p15, 0, Rd, c9, c6, 1    ; unlock data cache,
                             ; Rd ignored
```

——— **Note** ———

Prior to ARMv6, some format D implementations used c1 and c2 rather than c5 and c6. The technical reference manuals of implementations of architecture variants before ARMv6 must be checked, as provision of CP15 functionality was not mandated, and acted as a guideline only.

There are three rules about how many entries within a cache set can be locked:

• At least one entry per cache set must be left for normal cache operation. Failure to adhere to this restriction results in UNPREDICTABLE behavior.

• It is IMPLEMENTATION DEFINED how many ways in each cache set may be locked. MAX_CACHESET_ENTRIES_LOCKED < NSETS.

• It is IMPLEMENTATION DEFINED whether attempts to lock additional entries in format D are allocated as an unlocked entry or ignored.

For the instruction cache, a prefetch operation is used to fetch and lock individual cache lines. Each cache line is specified by its modified virtual address (see section B_TBD_FCSE_MVA). To lock code into the instruction cache, the following rules apply:

- the routine used to lock lines into the instruction cache must be executed from non-cacheable memory
- the code being locked into the instruction cache must be cacheable
- the instruction cache must be enabled and invalidated before locking down cache lines.

Failure to adhere to these restrictions causes UNPREDICTABLE results. Entries must be unlocked using the global instruction cache unlock command.

Cache lines must be locked into the data cache by first setting a global lock control bit. Data cache line fills occurring while the global lock control bit is set are locked into the data cache. To lock data into the data cache, the following rules apply:

- The data being locked must not exist in the cache. Cache clean and invalidate operations might be necessary to meet this condition.

- The data to be locked must be cacheable.

- The data cache must be enabled.

Failure to adhere to these restrictions causes UNPREDICTABLE results. Entries must be unlocked using the global instruction cache unlock command.

The Data Cache Lock register has the format shown in Table B6-15.

Table B6-15 Data cache lock register

31	1	0
SBZ/UNP		L
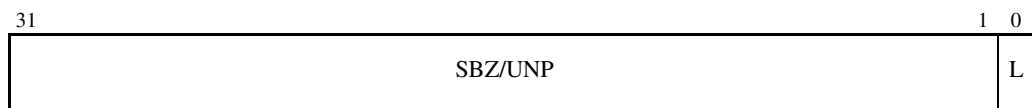
L (bit[0])	Lock bit	
	0	no locking occurs
	1	all data fills are locked while this bit is set.

Interactions with register 7 operations

Cache lockdown only prevents the normal replacement strategy used on cache misses from choosing to re-allocate cache lines in the locked-down region. Register 7 operations that invalidate, clean, or clean and invalidate cache contents affect locked-down cache lines as normal. If invalidate operations are used, you must ensure that they do not use virtual addresses or cache set/way combinations that affect the locked-down cache lines. (Otherwise, if it is difficult to avoid affecting the locked-down cache lines, repeat the cache lockdown procedure afterwards.)

Also, on some implementations and under some circumstances, the *clean single Dcache line* and *clean and invalidate single Dcache line* register 7 operations (see Table B6-6 on page B6-20) that identify a cache line by set/way can disrupt the constraints on the normal replacement strategy. To avoid this affecting the

locked-down area of the cache for Formats A, B and C, read the relevant CP15 register 9 (see section B_TBD_CWB_formatABCinstr) immediately before code that uses these operations, and write the value read back to the register immediately afterwards.

Chapter B7
Tightly Coupled Memory

This chapter describes *Tightly Coupled Memory* (TCM). It contains the following sections:

B7.1 About TCM

The TCM is designed to provide low latency memory that can be used by the processor without the unpredictability that is a feature of caches. Such memory can be used to hold critical routines, such as interrupt handling routines or real-time tasks where the indeterminacy of a cache would be highly undesirable. In addition, it can be used to hold scratchpad data, data types whose locality properties are not well suited to caching, and critical data structures such as interrupt stacks.

Up to four banks of data TCM and up to four banks of instruction TCM are supported by the architecture. Each bank must be programmed to be in a different location in the physical memory map.

The TCM is expected to be used as part of the physical memory map of the system, and is not expected to be backed by a level of external memory with the same physical addresses. For this reason, the TCM behaves differently from the caches for regions of memory that are marked as being Write-Through cacheable. In such regions, no external writes occur in the event of a write to memory locations contained in the TCM. There is an optional smartcache mode of operation where the TCM does adopt cache behavior over the prescribed (base address, size) memory region as defined in *SmartCache Behavior* on page B7-5.

Data and instructions can be transferred into and out of the TCMs with the L1 DMA described in *L1 DMA control using CP15 Register 11* on page B7-9.

Particular memory locations must be contained either in the TCM or in the cache. They must not be in both. In particular, no coherency mechanisms are supported between the TCM and the cache. This means that it is important when allocating the base address of the TCM to ensure that the same address ranges are not contained in the Cache.

B7.1.1 Restriction on Page Table Mappings

The TCM must appear to be implemented in a Physically Indexed, Physically Addressed manner, giving the following behaviors:

* The entries in the TCM do not need to be cleaned and/or invalidated by software for different virtual to physical mappings.

* Aliases to the same physical address can exist in memory regions that are held in the TCM. As a result the page mapping restrictions for the TCM are less restrictive than for the Cache.

B7.1.2 Restriction on Page Table Attributes

The page table entries that describe areas of memory that are handled by the TCM can be described as being Cacheable or Non-Cacheable, but must not be marked as Shared. If they are marked as either Device or Strongly Ordered, or have the Shared Attribute set, the locations that are contained within the TCM are treated as being Non-Shared, Non-Cacheable.

B7.2 TCM configuration and control

The System Control Coprocessor (CP15 registers 0, 1, and 9) are used for configuration and control of the TCMs in a system. Prior to ARMv6 it is IMPLEMENTATION DEFINED how TCMs are supported, though generally this is through a System Control Coprocessor interface.

B7.2.1 TCM Status Register CP15 Register 0

The number of TCMs that are implemented is implementation specific, and is identified by the TCM Status Register. This read-only register is accessed by reading CP15 register 0 with the opcode_2 field set to 2, as follows:

```
MRC p15, 0, Rd, C0, C0, 2 ; returns the TCM Status register
```

The format of the TCM Status Register is as shown:

31	29 28		19 18	16 15		3 2	0
0 0 0	SBZ/UNP		DTCM	SBZ/UNP		ITCM	

ITCM (Bits[2:0])

Indicate the number of Instruction (or Unified) TCMs implemented. This value lies in the range 0-4. All other values are reserved. All Instruction TCMs must be accessible to both instruction and data sides.

DTCM (Bits[18:16])

Indicate the number of Data TCMs implemented. This value lies in

the range 0-4. All other values are reserved.

B7.2.2 TCM Control bits in CP15 Register 1

The following bits in register 1 of the System Control Coprocessor have previously been used to control the TCM:

DT (bit[16])   This bit is now *Should be One* (SBO). This bit is used in the ARM946 and ARM966 cores to enable the Data TCM. In ARMv6, the TCM blocks have individual enables that apply to each block. As a result, the global bit is now redundant.

IT (bit[18])   This bit is now SBO. This bit is used in the ARM946 and ARM966 cores to enable the Instruction TCM. In ARMv6, the TCM blocks have individual enables that apply to each block. As a result, the global bit is now redundant.

B7.2.3 TCM Region Registers using CP15 Register 9

Each TCM bank has its own Region register. This register describes the physical base address and size of that TCM, and controls its enabling and mode of operation. Changing the TCM Region Register while a prefetch range or DMA operation is running has UNPREDICTABLE effects.

To access each of the TCM region registers, the TCM Selection register is set to the TCM of interest. These registers are accessible only in a privileged mode of operation, using CP15 register 9, as shown in Table B7-1.

Table B7-1 TCM Registers

ARM Instruction	TCM Region Register
`MRC/MCR P15, 0, Rd, C9, C1, 0`	Data TCM Region Register
`MRC/MCR P15, 0, Rd, C9, C1, 1`	Instruction/Unified TCM Region Register
`MRC/MCR P15, 0, Rd, C9, C2, 0`	TCM Selection Register

TCM selection register

The format of the TCM selection register is:

31	1	0
SBZ/UNP		TCM No.
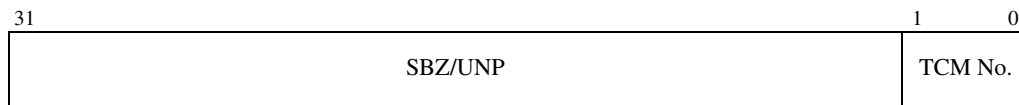
TCM Number (Bits[1:0])

> Indicates which TCM number the Region Registers apply to. This value is reset to 0. If the TCM Selection Register is written to point to a memory that is not implemented, then that write is IGNORED.

TCM region registers

The format of the TCM region registers is:

31	12 11	7 6	2	1	0
Base Address (Physical Address)	SBZ/UNP	Size		SC	En
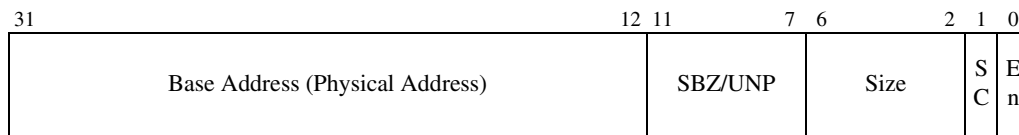
BaseAddress(Bits[31:12])

> Gives the Physical Base Address of the TCM. The Base Address is assumed to be aligned to the size of the TCM. Any bits in the range [(log2(RAMSize)-1):12] are ignored. The Base Address is 0 at Reset.

Size(Bits[6:2])

> On Reads, indicates the size of the TCM, and is IGNORED on writes. The encoding of the size field is shown in Table B7-2 on page B7-5.

SC (Bit[1])   This TCM is enabled as SmartCache (if the TCM supports SmartCache):

0 = Local RAM (reset state)

1 = SmartCache

If the RAM does not support SmartCache, the bit is read as zero, and is ignored on writes. This can be used to determine if the TCM supports SmartCache.

En (Bit[0])   0 = Disabled (Reset state)

1 = Enabled

If a TCM is not implemented, then the TCM Region register for that TCM is SBZ/UNP.

The implementation-specific representation of the sizes is shown in Table B7-2.

Table B7-2 TCM sizes

Size field	Memory size	Size field	Memory size
0b00000	0K	0b01101	4M
0b00011	4K	0b01110	8M
0b00100	8K	0b01111	16M
0b00101	16K	0b10000	32M
0b00110	32K	0b10001	64M
0b00111	64K	0b10010	128M
0b01000	128K	0b10011	256M
0b01001	256K	0b10100	512M
0b01010	512K	0b10101	1G
0b01011	1M	0b10110	2G
0b01100	2M	0b10111	4G

The base address of each TCM must be different, such that no location in memory is contained in more than one TCM. If a location in memory is contained in more than one TCM, it is UNPREDICTABLE which memory the instruction or data is returned from. Implementations must ensure that this situation cannot result in physical damage to the TCM.

B7.2.4   SmartCache Behavior

When a TCM is configured as SmartCache, it forms a contiguous area of cache, with the contents of memory backed by external memory. Each line of the TCM is the same length as the cache line (indicated in the Cache Type Register for the equivalent Cache), and can be individually set as being valid or invalid. Writing

the RAM Region Register causes the valid information for each line to be cleared (marked as invalid). When a read access is made to an invalid line, the line is fetched from the L2 memory system in exactly the same way as for a cache miss, and the fetched line is then marked as valid.

The number of TCMs that support SmartCache is IMPLEMENTATION DEFINED.

For the TCM to exhibit SmartCache behavior, areas of memory that are covered by a TCM operating as SmartCache must be marked as Cacheable. For a memory access to a memory location that is marked as Non-Cacheable but is in an area covered by a TCM, if the corresponding SmartCache line is marked as Invalid, then the memory access must not cause the location to be fetched from external memory and marked as valid. If the corresponding SmartCache line is marked as Valid, then it is UNPREDICTABLE whether the access is made to the TCM or to External memory.

Areas of memory that are marked as Shared can only be covered by the SmartCache if the implementation supports mechanisms that make the SmartCache transparent to memory. It is therefore IMPLEMENTATION DEFINED whether regions of memory covered by the SmartCache can be marked as Shared.

B7.2.5 Local RAM Behavior

When a TCM is configured as Local RAM, then it forms a continuous area of memory that is always valid if the TCM is enabled. It therefore does not use the valid bits for each line that are used for SmartCache. The TCM configured as Local RAM is expected to be used as part of the physical memory map of the system, and is not expected to be backed by a level of external memory with the same physical addresses. For this reason, the TCM behaves differently from the caches for regions of memory that are marked as being Write-Through Cacheable. In such regions, no external writes occur in the event of a write to memory locations contained in the TCM.

The DMA can only operate to an area of TCM that is configured as Local RAM. This avoids any requirement for interactions between the cache refill and DMA operations. Attempting to perform a DMA to an area of TCM that is configured as SmartCache results in an internal DMA error (Tightly-Coupled DMA out of range) as described in *L1 DMA control using CP15 Register 11* on page B7-9.

B7.3 Accesses to TCM and cache

In the event that a TCM and a cache both contain the requested address, it is UNPREDICTABLE which memory the instruction or data is returned from. Implementations must ensure that this situation cannot result in physical damage to the cache or TCM. It is expected that such an event should only arise from a failure to invalidate the cache when the base register of the TCM is changed. This is a programming error.

For a Harvard arrangement of caches and TCM, it is required that the Data Reads and Writes can access any Instruction TCM configured as Local Memory for both reads and writes. This ensures that accesses to literal pools, undefined instructions, and SWI numbers are possible, and facilitates debugging. The arbitration between the ports of such an implementation is IMPLEMENTATION DEFINED. For this reason, an instruction TCM configured as Local Memory must behave as a unified TCM, but can be optimized for instruction fetches. This requirement only exists for the TCMs when configured as Local RAM.

An instruction memory barrier must be inserted between a write to an Instruction TCM and the instructions being written being relied on. In addition, any branch prediction mechanism should be invalidated or disabled if a branch in the Instruction TCM is overwritten.

The converse arrangement, that instruction port(s) can access the Data TCM, is not required. An attempt to access addresses in the range covered by a Data TCM from an instruction port does not result in an access to the Data TCM. In this case, the instruction must be fetched from main memory. Such accesses might result in external aborts in some systems, because the address range might not be supported in main memory.

An Instruction TCM must not be programmed to the same base address as a Data TCM (and in the event of different sizes of the two RAMs, the regions in physical memory of the two RAMs must not be overlapped), unless each TCM is configured to operate as SmartCache. If a Data and an Instruction TCM overlap, and either is not configured as SmartCache, it is UNPREDICTABLE which memory the instruction data is returned from. Implementations must ensure that this cannot result in physical damage to the TCM.

B7.4   Level 1 (L1) DMA model

The purpose of the L1 DMA is to provide a background route to transfer blocks of data to or from the TCMs. Its main emphasis is on relatively large blocks of data being moved, rather than individual words or small structures. The number of DMA channels that can be implemented is IMPLEMENTATION DEFINED and can be 0.

The L1 DMA can be initiated and controlled by accessing the appropriate System Control Coprocessor (CP15) registers and instructions. The process specifies the internal start and end addresses and external start address, together with the direction of the DMA. The addresses specified are virtual addresses, and the L1 DMA hardware must include translation of virtual addresses to physical addresses and checking of protection attributes. Implementations can use the TLB tbd xref to hold the page table entries for the DMA, but must ensure that the entries in a TLB used by the DMA are consistent with the page tables. Errors, arising from protection checks, can be configured to signal the CPU using an interrupt.

Completion of the DMA can also be configured to signal the CPU with an interrupt using the same interrupt to the CPU that the error uses.

The status of the DMA can be read from the CP15 registers associated with the DMA.

An implementation-defined number of DMA channels are available, each with their own set of control and status registers. The maximum number of DMA channels that can be defined is architecturally limited to 2. Only 1 DMA channel can be active at a time. If the other DMA channel has been started, the newly activated channel is queued to start performing memory operations after the currently active channel has completed.

DMA transfers must be between external memory and TCM, in the specified direction. Transfers between the Instruction TCM and the Data TCM are not supported, nor are transfers between two memory locations outside L1. Attempts to perform such transfers result in an error being reported by the DMA channel.

The L1 DMA behaves as a distinct master from the rest of the CPU, and the same mechanisms for handling shared memory regions must be used if the external addresses being accessed by the L1 DMA system are also accessed by the rest of the CPU. These are discussed in tbd xref section 6. If a User mode DMA transfer is performed using an external address that is not marked as Shared, an error is signalled by the DMA channel.

There is no ordering requirement of memory accesses caused by the L1 DMA relative to those generated by reads and writes by the CPU, while a channel is running. When a channel has completed running, all its transactions are visible to all other observers in the system. All memory accesses caused by the DMA occur in the order specified by the DMA channel, regardless of the memory type.

If a DMA is performed to Strongly Ordered memory (tbd xref see section 6.5), a transaction caused by the DMA prevents any further transactions being generated by the DMA until the access is complete. A transaction is complete when it has changed the state of the target location or data has been returned to the DMA.

If the FCSE PID, the Domain Access Control register, or the page table mappings are changed, or the TLB flushed, while a DMA channel is in the Running or Queued state, then it is UNPREDICTABLE when the effect of these changes is seen by the DMA.

B7.5 L1 DMA control using CP15 Register 11

The L1 DMA is controlled using CP15 Register 11. There are several registers associated with CP15 Register 11, that are used for the control and monitoring of the DMA. They are defined in Table B7-3. The instructions are:

```
MCR p15, 0, Rd, c11, CRm, Opcode2
MRC p15, 0, Rd, c11, CRm, Opcode2
```

where `CRm` is the associated register, and `Rd` is the ARM source or destination register. These, and `Opcode2`, are as shown in Table B7-3. Further details are given in the following subsections.

Table B7-3 L1 DMA Control Registers

Register	CRm	Opcode 2	Function
Identification/Status	0	Present/Queued/Running/Interrupting	Privileged only: Read-only
User Accessibility	1	0	Privileged only: Read/Write
Channel Number	2	0	Read/Write
Enable	3 [a]	Stop/Start/Clear	Write-only
Control	4 [a]	0	Read/Write
Internal Start Address	5 [a]	0	Read/Write
External Start Address	6 [a]	0	Read/Write
Internal End Address	7 [a]	0	Read/Write
Channel Status	8 [a]	0	Read-only
RESERVED SBZ/UNP	9-14	0	Read/Write
Context ID	15 [a]	0	Privileged only: Read/Write

a. One register per channel.

The Enable, Control, Internal Start Address, External Start Address, Internal End Address, Channel Status, and Context ID registers are multiple registers, with one register of each existing for each channel that is implemented. Which register is accessed is determined by the Channel Number register, as described in *TCM Region Registers using CP15 Register 9* on page B7-3.

B7.5.1 User Access to Cp15 Register 11 operations

A number of CP15 Register 11 operations can be executed by code while in User mode.

Attempting to execute a privileged operation in User mode using CP15 register 11 results in an Undefined Instruction Exception.

B7.5.2   Identification and Status Registers

These registers define the DMA Channels that are physically implemented on the particular device and their current status. They can be used by processes handling DMA to determine the physical resources implemented and their availability. The bottom two bits, the channel bits, of each register correspond to the 2 channels that are defined architecturally (bit 0 corresponding to channel 0, bit 1 to channel 1).

The Opcode2 value distinguishes the registers implemented as shown in Table B7-4.

Table B7-4 L1 DMA Identification and Status registers

Opcode2	Function
0	Present = 1 for a present channel, 0 for an absent channel.
1	Queued = 1 for a channel that is Queued, 0 otherwise. Unimplemented channels return 0.
2	Running = 1 for a channel that is Running, 0 otherwise. Unimplemented channels return 0.
3	Interrupting = 1 for a channel that is causing an interrupt (through completion or an error), 0 otherwise. Unimplemented channels return 0.
4-7	Reserved, UNPREDICTABLE.

These registers can only be read by a privileged process. Attempting to access them by a user process results in an Undefined Instruction exception.

Registers 0-3 have the format shown:

31 2	1 0
UNP	Channel bits
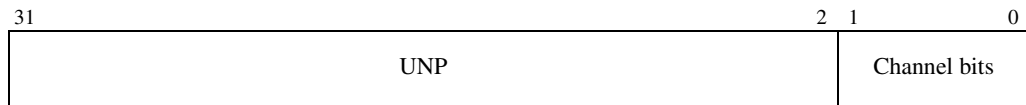
The instruction to access these registers is:

```
MRC p15, 0, Rd, c11, c0, n ; where n is 0, 1, 2, or 3
```

B7.5.3   User Accessibility Register

This register contains a bit of each channel, referred to as the U bit for that channel, that indicates whether the registers for that channel can be accessed by a user mode process. The registers that can be accessed if the U bit for that channel is 1 are:

- Enable
- Control
- Internal Start Address
- External Start Address
- Internal End Address
- Channel Status.

The contents of these registers must be preserved on a task switch if the registers are user accessible.

If the U bit for that channel is set to 0, then attempting to access them by a user process results in an Undefined Instruction Exception.

The user accessibility register has the format shown:

31	2	1	0
UNP		Channel bits	
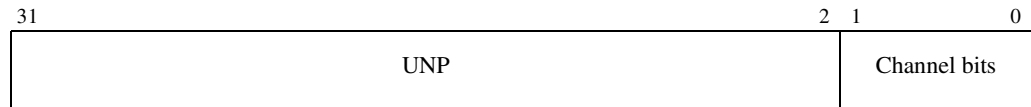
The instructions to access these registers are:

```
MCR p15, 0, Rd, c11, c1, 0
MRC p15, 0, Rd, c11, c1, 0
```

B7.5.4   Channel Number Register

The Enable, Control, Internal Start Address, External Start Address, Internal End Address, Channel Status, and Context ID registers are multiple registers, with one register of each existing for each channel that is implemented. The value contained in the channel number register is used to determine which of the multiple registers is accessed when one of these registers is specified.

This register can be accessed by user processes if the U Bit of any channel is set to 1. Attempting to access them by a user process if no channel has the U bit set to 1 results in an Undefined Instruction Exception.

The DMA Channel Number Register has the format:

31	1	0
UNP		Channel No.
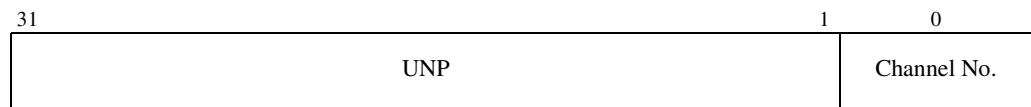
The instructions to access these registers are:

```
MCR p15, 0, Rd, c11, c2, 0
MRC p15, 0, Rd, c11, c2, 0
```

B7.5.5   Enable Registers

Each implemented DMA channel has its own register location that can be written to start, stop, or clear a channel. The value of Opcode2 in the MCR instruction determines the operation to be performed, as shown in Table B7-5 on page B7-12.

The instruction to access these registers is:

```
MCR p15, 0, Rd, c11, c3, n ; where n is 0, 1, or 2
```

Table B7-5 DMA Channel Enable operations

Opcode2	Operation
0	Stop
1	Start
2	Clear
3-7	Reserved

If the U bit for a channel is set to 1, a user process can perform these operations for that channel. If the U bit for the channel is set to 0 and a user process attempts to perform one of these operations, the result is an Undefined Instruction Exception.

The channel status is described in *Channel Status Registers* on page B7-16.

Start

The Start command causes the channel to begin doing DMA transfers. The channel status is changed to Running on the execution of a Start command if the other DMA channel is not in operation at that time. Otherwise it is set to Queued. A channel is in operation if its status is Queued or Running or if the channel is indicating an Error, that is, it has a Status of Error/Completed with either the Internal or External error value greater than or equal to 0b01000.

Stop

The Stop command is issued when the channel status is Running. The DMA channel ceases to do memory accesses as soon as possible after the issuing of the instruction. For accesses to restartable external memory, this can be accelerated by abandoning accesses that have been read from external memory but not yet written in the TCM. This acceleration approach cannot be used for DMA transactions to or from memory regions marked as Device.

The DMA channel can take a number of cycles to stop after issuing a Stop instruction. The channel status remains at Running until the channel has stopped. The channel status is set to Idle at the point that all outstanding memory accesses have completed. The start address registers contain the addresses required to restart the operation when the channel has stopped.

If the Stop command is issued when the channel status is Queued, the channel status is changed to Idle.

The Stop has no effect if the channel status is not Running or Queued.

Clear

The Clear command causes the channel status to change from Error/Completed to Idle. It also clears the Interrupt that is set by the channel as a result of an error or completion (as defined in the control register in tbd xref). The contents of the Internal and External Start Address registers are unchanged by this command.

Performing a Clear command when the channel has the status of Running or Queued has no effect.

Debug implications for the DMA

The L1 DMA behaves as a separate engine from the processor core, and when started works autonomously. As a result, if the L1 DMA has channels with the status of Running or Queued, these channels continue to run, or can start running, even if the processor is stopped by debug mechanisms. This can result in the contents of the TCM changing while the processor is stopped in Debug. The DMA channels must be stopped by a Stop operation to avoid this.

B7.5.6 Control Registers

Each implemented DMA channel has its own register for control of the DMA operation. The register format for these is:

31	30	29	28	27	26	25	20	19	8	7	2	1	0
TR	DT	IC	IE	FT	UM	UNP/SBZ		ST		UNP/SBZ		TS	
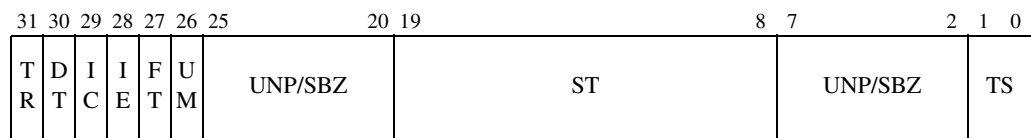
The instructions to access these registers are:

```
MCR p15, 0, Rd, c11, c4, 0
MRC p15, 0, Rd, c11, c4, 0
```

TS (Bits[1:0]) Transaction Size. The transaction size denotes the size of the transactions performed by the DMA channel. This is particularly important for Device or Strongly Ordered memory locations because it ensures that accesses to such memory occur at their programmed size.

00 = Byte

01 = Halfword

10 = Word

11 = Double Word (8 bytes).

ST (Bits[19:8])

Stride (in bytes). The Stride indicates the increment on the external address between each consecutive access of the DMA. A Stride of zero indicates that the external address should not be incremented. This is designed to facilitate the accessing of volatile locations such as a FIFO.

The value of the stride must be aligned to the Transaction Size, otherwise this can result in UNPREDICTABLE behavior.

The Stride is interpreted as a positive number or zero.

The internal address increment is not affected by the stride, but is fixed at the transaction size.

UM (Bit[26]) User Mode. Indicates that the permission checks are based on the DMA being in User mode or privileged mode:

0 = Transfer is a privileged transfer.

1 = Transfer is a user mode transfer.

If the U bit is set for the channel, then the UM bit can only be written as 1. Attempting to write the value 0 for this bit in this case has no effect.

FT (Bit[27]) Full Transfer. This indicates that the DMA transfers all words of data as part of the DMA that is transferring data from the TCM to the External memory:

0 = Transfer at least those locations in the address range of the DMA in the TCM that have been changed by a store operation since the location was written to or read from by an earlier DMA that had the FT bit equal to 0, or since reset, whichever is the more recent operation. Implementations are expected to minimize the number of transfers from the TCM as a result of this bit value

1 = Transfer all locations in the address range of the DMA, regardless of whether or not the locations have been changed by a store. An access by the DMA to the TCM with the FT bit equal to 1 does not cause the record of what locations have been written to be changed.

IE (Bit[28]) Interrupt on Error. The action of this bit depends on the setting of the U bit (see *User Accessibility Register* on page B7-10):

If U = 0 and IE[28] = 0, the DMA channel does not assert an Interrupt on Error.

If either U = 1 or IE[28] = 1, the DMA channel asserts an interrupt on an error.

The interrupt is de-asserted (from this source) on the channel being set to Running with a Start operation (see *Enable Registers* on page B7-11), or to Idle with a Clear operation. All DMA transactions on channels that have the U bit set to 1 (see *User Accessibility Register* on page B7-10) interrupt on error regardless of the value written to this bit.

IC (Bit[29]) Interrupt on Completion. The interrupt on completion bit indicates that the DMA channel should assert an interrupt on completing the DMA transfer. The interrupt is de-asserted (from this source) if the clear operation is performed on the channel causing the interrupt (see *Enable Registers* on page B7-11). The U bit has no effect on whether an interrupt is generated on completion.

0 = No Interrupt on Completion

1 = Interrupt on Completion.

DT (Bit[30]) Direction of transfer:

0 = from L2 memory to the TCM

1 = from the TCM to the L2 memory.

TR (Bit31) Target TCM:

ARM DDI 0100F-02

0 = Data (or unified)

1 = Instruction.

If the U bit for the channel is set to 0, attempting to access the register by a user process results in an Undefined Instruction Exception. Attempting to write to the Control register while the channel has the status of Running or Queued results in UNPREDICTABLE effects.

Implementation Note

The mechanism for implementing the functionality of the FT bit equal to 0 is IMPLEMENTATION DEFINED. The marking of each line (or multiple lines) in the TCM with *dirty bits* that record that a store to a location within that line has occurred (that is, that the line is dirty) is an acceptable implementation, even though it might result in some locations being incorrectly marked as dirty. Such implementations must not mark locations as clean within a TCM line, that are not part of the DMA transfer, as a result of a DMA transfer. In the case of a DMA write from the TCM (DT == 1), the dirty bits for a line would be cleared if the entire line were written by the DMA transfer, but the dirty bits would be unchanged if the DMA transfer is only writing part of the line.

B7.5.7 Internal Start Address Registers

These registers define the first address in the TCM for each DMA channel, that is the first address to or from which the data is to be transferred. The Internal Start Address is a virtual address, whose physical mapping should be described in the page tables at the time that the channel is started. The memory attributes for that virtual address are used in the transfer. Memory permission faults can be generated. The Internal Start Address must lie within a TCM, otherwise an error is reported in the Channel Status register. The marking of memory locations in the TCM as being Device results in UNPREDICTABLE effects.

The contents of this register are UNPREDICTABLE while the DMA channel is Running. When the channel is stopped because of a Stop command, or an error, it contains the address required to restart the transaction. On completion, it contains the address equal to the Internal End Address.

The Internal Start Address must be aligned to the transaction size set in the control register otherwise the effects are UNPREDICTABLE.

If the U bit for the channel is set to 0, then attempting to access the register by a user process results in an Undefined Instruction Exception. Attempting to write this register while the DMA channel is Running or Queued has no effect (that is, it fails without issuing an error).

To read and write this register, use the following instructions:

```
MCR p15, 0, Rd, c11, c5, 0
MRC p15, 0, Rd, c11, c5, 0
```

B7.5.8 External Start Address Registers

These registers define the first address in external memory for each DMA channel, that is the first address to or from which the data is to be transferred. The External Start Address is a virtual address, whose physical mapping should be described in the page tables at the time that the channel is started. The memory attributes for that virtual address are used in the transfer. Memory permission faults can be generated.

The External Start Address must lie in the external memory beyond the L1 memory system otherwise the results are UNPREDICTABLE.

The contents of this register are UNPREDICTABLE while the DMA channel is Running. When the channel is stopped because of a Stop command or an error, it contains the address required to restart the transaction. On completion, it contains the address equal to the final address that was accessed plus the Stride.

The External Start Address must be aligned to the transaction size set in the control register, otherwise the effects are UNPREDICTABLE.

If the U bit for the channel is set to 0, then attempting to access the register by a user process result in an Undefined Instruction Exception. Attempting to write this register while the DMA channel is Running or Queued has no effect.

To read and write this register, use the following instructions:

```
MCR p15, 0, Rd, c11, c6, 0
MRC p15, 0, Rd, c11, c6, 0
```

B7.5.9   Internal End Address Registers

These registers define the internal end address. The value set must be greater than the internal start address. The internal end address is the final internal address (modulo the transaction size) that the DMA accesses, plus the transaction size. The internal end address is the first (incremented) address that the DMA does not access. When the transaction associated with the final internal address has completed, the whole DMA transfer is complete.

The Internal End Address must be aligned to the transaction size set in the control register, otherwise the effects are UNPREDICTABLE.

If the U bit for the channel is set to 0, then attempting to access the register by a user process results in an Undefined Instruction Exception. Attempting to write to this register while the DMA channel is Running or Queued has no effect.

To read and write this register, use the following instructions:

```
MCR p15, 0, Rd, c11, c7, 0
MRC p15, 0, Rd, c11, c7, 0
```

B7.5.10   Channel Status Registers

These registers define, for each channel, the status of the most recently started DMA operation on that channel. It is a read-only register. The format of the channel status register is:

31 12	11 7	6 2	1 0
UNP/SBZ	ES	IS	Status
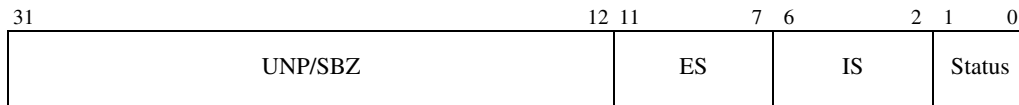
To read this register, use the following instruction:

```
MRC p15, 0, Rd, c11, c8, 0
```

Status (bits[1:0])

  00 = Idle

  01 = Queued

  10 = Running

  11 = Complete/Error.

IS (bits[6:2]) Internal Address Error Status, with the following encoding:

  0b00xxx = No Error (reset value)

  0b01000 = TCM out of range

  0b11100 = External Abort on Translation of 1st Level Page Table

  0b11110 = External Abort on Translation of 2nd Level Page Table

  0b10101 = Translation fault (Section)

  0b10111 = Translation fault (Page)

  0b11001 = Domain fault (Section)

  0b11011 = Domain fault (Page)

  0b11101 = Permission fault (Section)

  0b11111 = Permission fault (Page).

  All other encodings are Reserved.

ES (bits[11:7])

  External Address Error Status, with the following encoding:

  0b00xxx = No Error (reset value)

  0b01001 = Unshared Data Error

  0b11010 = External Abort (can be imprecise)

  0b11100 = External Abort on Translation of 1st Level Page Table

  0b11110 = External Abort on Translation of 2nd Level Page Table

  0b10101 = Translation fault (Section)

  0b10111 = Translation fault (Page)

  0b11001 = Domain fault (Section)

  0b11011 = Domain fault (Page)

  0b11101 = Permission fault (Section).

  0b11111 = Permission fault (Page).

  All other encodings are Reserved.

If an error occurs, the faulting address is contained in the appropriate start address register, unless the error is an External Error (ES == 0b11010).

A channel with the state of Queued changes to Running automatically if the other channel, if implemented, changes to Idle or Complete/Error with no error.

When a channel has completed all of the transfers of the DMA, so that all changes to memory locations caused by those transfers are visible to other observers, its status changes from Running to Complete/Error. This change does not happen before the external accesses from the transfer have completed.

If the U bit for the channel is set to 0, then attempting to read the register by a user process results in an Undefined Instruction Exception.

The Unshared Data External Address Error is signalled if a DMA transfer, that has the UM bit set in the Control register, attempts to access external memory locations if those memory locations are not marked as Shared. If the UM bit is clear, this error cannot occur.

B7.5.11 Context ID Registers

This register contains, for each implemented DMA channel, the Context ID Register of the process that is using the channel. It must be written with the CPU Context ID of the process that uses the channel as part of the initialization of that channel. Where the channel is being designated as a user accessible channel, the Context ID is written by the privileged process that initializes the channel for user usage, that is, at approximately the same time that the U bit for the channel is written. The bottom eight bits of the Context ID Register are used in the address translation from virtual to physical addresses to allow different virtual address maps to co-exist. Attempting to write this register while the DMA channel is Running or Queued has no effect.

This register can only be read by a privileged process to provide anonymity of the DMA channel usage from user processes. It can only be written by a privileged process for security reasons. On a context switch, where the state of the DMA is being stacked and restored, this register should be included in the saved state.

The format of this register is:

31	0
Context ID	
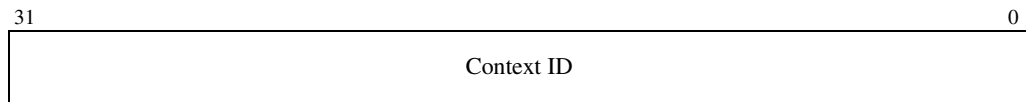
Attempting to access this privileged register by a user process results in an Undefined Instruction Exception.

To read and write this register, use the following instructions:

```
MCR p15, 0, Rd, c11, c15, 0
MRC p15, 0, Rd, c11, c15, 0
```

——— **Note** ———

A DMA channel and the associated ContextID register use the currently active pagetable. Software must ensure that the is no active DMA, either Running or Queued, that can be affected by page table updates.

Chapter B8
Fast Context Switch Extension

This chapter describes the *Fast Context Switch Extension* (FCSE). It contains the following sections:

- *About the FCSE* on page B8-2
- *Modified virtual addresses* on page B8-3
- *Enabling the FCSE* on page B8-5
- *CP15 registers* on page B8-6.

B8.1 About the FCSE

The *Fast Context Switch Extension* (FCSE) modifies the behavior of an ARM memory system. This modification allows multiple programs running on the ARM processor to use identical address ranges, while ensuring that the addresses they present to the rest of the memory system differ.

Normally, a swap between two software processes whose address ranges overlap requires changes to be made to the virtual-to-physical address mapping defined by the MMU's page tables (see Chapter B4 *Virtual Memory System Architecture*). It also typically causes cache and TLB contents to become invalid (because they relate to the old virtual-to-physical address mapping), and so requires caches and TLBs to be flushed. As a result, each process swap has a considerable overhead, both directly because of the cost of changing the page tables and indirectly because of the cost of subsequently reloading caches and TLBs.

By presenting different addresses to the rest of the memory system for different software processes even when they are using identical addresses, the FCSE avoids this overhead. It also allows software processes to use identical address ranges even when the rest of the memory system does not support virtual-to-physical address mapping.

——— **Note** ———

The FCSE mechanism is deprecated in ARMv6

B8.2   Modified virtual addresses

The 4GB virtual address space is divided into 128 process blocks, each of size 32MB. Each process block can contain a program which has been compiled to use the address range `0x00000000` to `0x01FFFFFF`. For each of i=0 to 127, process block i runs from address (i× `0x02000000`) to address (i× `0x02000000` + `0x01FFFFFF`).

The FCSE processes each virtual address for a memory access generated by the ARM processor to produce a *modified virtual address*, which is sent to the rest of the memory system to be used in place of the normal virtual address. For an MMU-based memory system, the process is illustrated in Figure B8-1:
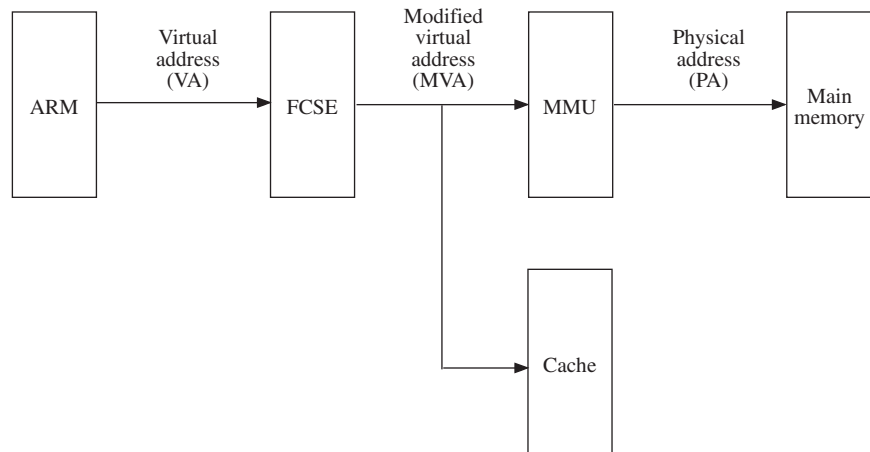
Figure B8-1 Address flow in MMU memory system with FCSE

When the ARM processor generates a memory access, the relationship between the *Virtual Address* (VA) and *Modified Virtual Address* (MVA) is:

```
if (VA[31:25] == 0b0000000) then
    MVA = VA | (PID << 25)
else
    MVA = VA
```

where `PID` is a 7-bit number that identifies which process block the current process is loaded into. This is also known as the *process ID* of the current process.

———— **Note** ————

Virtual addresses are sometimes passed to the memory system as data, as for example in some of the cache control operations described in tbd xref Register 7: Cache functions on page B15-15. For these operations, no address modification occurs, and MVA = VA.

———————————————

Copyright © 1996–2003 ARM Limited. All rights reserved.
Confidential - Draft

Each process is compiled to use the address range `0x00000000` to `0x01FFFFFF`. When referring to its own instructions and data, therefore, the program generates VAs whose top seven bits are all zero. The resulting MVAs have their top seven bits replaced by PID, and so lie in the process block of the current process.

The program is also allowed to generate VAs whose top 7 bits are not all zero. When this happens, the MVA is equal to the VA. This allows the program to address the process block of another process, provided the other process does not have process ID 0. Provided access permissions are set correctly, this can be used for inter-process communication.

───── **Note** ─────

It is recommended that only process IDs 1 and above are used for general-purpose processes, because the process with process ID 0 cannot be communicated with in this fashion.

Use of the FCSE therefore allows the cost of a process swap to be reduced to:

• The cost of a write of the PID.

• The cost of changing access permissions if they need changing for the new process. In an MMU-based system, this might require a TLB flush. However, this is usually significantly cheaper than the cache flush that would have been required without the FCSE. Furthermore, even the TLB flush can often be avoided by the use of domains. This reduces the cost to that of a write to the Domain Access Control Register (see tbd xref Domains on page B13-19).

B8.3 Enabling the FCSE

When PID == 0b0000000, the rules for processing a VA always result in MVA == VA, as if the FCSE were not present.

There is therefore no specific FCSE enable bit. Instead, the PID is initialized to 0b0000000 on reset, resulting in the FCSE being effectively disabled.

The FCSE can then be enabled by writing a non-zero value to the PID, and disabled by writing 0b0000000 to the PID.

B8.4 CP15 registers

The FCSE only uses one coprocessor 15 register, namely register 13.

B8.4.1 Register 13: Process ID

31	25 24	0
PID	UNP/SBZP	
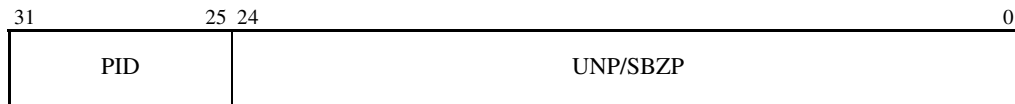
Reading register 13 returns the PID in bits[31:25]. Bits[24:0] of the value read are UNPREDICTABLE.

Writing register 13 sets the PID to bits[31:25] of the value written. Bits[24:0] of the value written should be zero or bits[24:0] of a value previously read from register 13. The results of writing any other value to bits[24:0] are UNPREDICTABLE.

In MCR and MRC instructions used to write and read register 13, <CRm> should be c0 and <opcode2> should be 0 (or omitted). If they have other values, the instruction is UNPREDICTABLE.

——— **Note** ———

When the PID is written, the overall virtual-to-physical address mapping changes. Because of this, care must be taken to ensure that instructions which might have already been prefetched are not affected by the address mapping change.

Formally, this means that an IMB sequence is required, as described in *Instruction Memory Barriers (IMBs)* on page B2-29. However, the mapping from the modified virtual address to the physical address is unchanged, so typically there is no need to flush caches, etc. This means that an implementation can usually specify a restricted form of the IMB sequence to be used after using an MCR to write to register 13.

Typically, this means that all instructions reached by predictable subsequent execution from the MCR instruction must be fetched with addresses whose top seven bits are not 0b0000000. For the definition of *predictable subsequent execution* and further details, see *Other uses for IMBs* on page B2-30.

Part C

Vector Floating-point Architecture

Chapter C1
Introduction to the Vector Floating-point Architecture

This chapter gives an introduction to the *Vector Floating-Point* (VFP) architecture, and its compliance with the IEEE 754 standard. It contains the following sections:

- *About the Vector Floating-point architecture* on page C1-2
- *Overview of the VFP architecture* on page C1-4
- *Compliance with the IEEE 754 standard* on page C1-8
- *IEEE 754 implementation choices* on page C1-9.

C1.1 About the Vector Floating-point architecture

The *Vector Floating-Point* (VFP) architecture is a coprocessor extension to the ARM architecture. It provides single-precision and double-precision floating-point arithmetic, as defined by *ANSI/IEEE Std. 754-1985 IEEE Standard for Binary Floating-Point Arithmetic*. This document is referred to as the *IEEE 754 standard* in the following text.

Short vectors of up to 8 single-precision or 4 double-precision numbers are handled particularly efficiently by the VFP architecture. Most arithmetic instructions can be used on these vectors, allowing *single-instruction, multiple-data* (SIMD) parallelism. Furthermore, the floating-point load and store instructions have *multiple register* forms, allowing vectors to be transferred to and from memory efficiently.

tbd still true? Double-precision support is optional, with its presence being indicated by the variant letter D. So the VFPv1D variant has both single-precision and double-precision, while VFPv1xD supports single-precision only. By default, double-precision support is present.

ARM introduced its VFP architecture with the second edition of the *ARM Architecture Reference Manual*. This version is known as VFPv1. It was implemented in ARM10 Rev0.

VFPv2 supersedes VFPv1. VFPv2 is the version used in all ARM10 derivatives except the rev0 implementation, in all ARM9 VFP coprocessors, and in all current developments.

An implementation of VFP can include a software component entered through the undefined instruction vector. This allows consistent software behavior with varying degrees of hardware support:

*   Software emulation of undefined instructions. That is, the remainder is not part of the VFPv2 architecture.

*   Software completion of instructions where necessary to meet correctness criteria for such requirements as rounding, overflow/underflow, handling NaNs, and denormals.

*   Error conditions such as invalid operations and divide by zero.

Software exceptions fall into two categories:

Traps   IEEE-defined exceptions that a hardware implementation must pass back to application software to resolve.

Bounces   Software entry points for task completion determined by an implementation. They might be used for rare conditions, operations that are difficult to implement in hardware, or operations that are gate intensive in hardware.

The architecture provides options that enable simpler arithmetic than defined by IEEE 754 for denormal and NaN values. Implementations choose the hardware/software divide. An implementation can choose to implement full IEEE 754 in hardware when traps are disabled. ARM implementations require a software component for full IEEE 754 compliance, only executing everything in hardware for the RunFast mode tbd xref.

Implementations use CP10 and CP11 for VFP instruction space. In general, CP10 is used to encode single-precision operations, and CP11 is used to encode double-precision operations. All unused codes are reserved.

The definition of the interface between the VFP hardware and the software component of the implementation is known as the *sub-architecture*. The intention is to provide a consistent interface for applications and operating system support. RunFast mode is an example of a sub-architecture feature.

C1.1.1 VFPv1 to VFPv2 changes

*   Bits[19:16] of the FPSID floating point identification register are 0b0001 for VFPv2. They were 0b0000 for VFPv1. See tbd xref for details of the FPSID register.

*   There are new coprocessor instruction equivalents of MRRC and MCRR for loading and storing a pair of ARM registers from and to the VFP coprocessor:

    FMDRR   Transfer two 32-bit ARM registers to a double-precision VFP register, see *FMDRR* on page C4-55.

    FMRRD   Transfer a double-precision VFP register to two 32-bit ARM registers, see *FMRRD* on page C4-58.

    FMSRR   Transfer two 32-bit ARM registers to a pair of single-precision VFP registers, see *FMSRR* on page C4-71.

    FMRRS   Transfer a pair of single-precision VFP registers to two 32-bit ARM registers, see *FMRRS* on page C4-59.

*   There are three new bits in the FPSCR register:

    DN bit   Enable default NaN mode when set. See *Default NaN mode* on page C2-14.

    IDE bit   Input denormal trap enable. See *Input denormal flag and trap enable* on page C2-25.

    IDC bit   Input denormal detected. In *Flush to Zero* (FTZ) mode this bit indicates inputs flushed to zero. This is a sticky bit. That is, when it is set, it remains set until it is cleared by an explicit write to the FPSCR register. See *Input denormal flag and trap enable* on page C2-25.

C1.1.2 Software Support

Three defined software models complement the ARM implementation of VFPv2:

Full IEEE 754   Supports all features and the concept of trap handlers. Opportunities to optimize code are very limited in this model.

Java Mode   A subset of IEEE 754 behavior as defined by the Java standard. There are no exception traps. Only one rounding mode supported.

This mode permits code motion optimizations, for example CSE, PRE, and loop-invariant hoisting, but is strict about optimizing expressions.

RunFast mode   Invoked when FTZ and DN modes are enabled, and all exceptions disabled. All conditions adopt a default condition in hardware making execution fully deterministic. This is not IEEE compliant. This model permits a wide variety of floating-point optimizations.

C1.2 Overview of the VFP architecture

This section provides a brief overview of the VFP architecture. More extensive and detailed information on the architecture is given in Chapter C2 *VFP Programmer's Model*.

C1.2.1 Registers

VFP has 32 general-purpose registers, each capable of holding a single-precision floating-point number or a 32-bit integer. In D variants of the architecture, these registers can also be used in pairs to hold up to 16 double-precision floating-point numbers. There are also three or more system registers:

FPSID   Is read-only. It can be read to determine which implementation of the VFP architecture is being used.

FPSCR   Supplies all user-level status and control. Status bits hold comparison results and cumulative flags for floating-point exceptions. Control bits are provided to select rounding options and vector length/stride, and to enable floating-point exception traps.

FPEXC   Contains a few bits for system-level status and control.

The remaining bits of the FPEXC register and any further system registers are IMPLEMENTATION DEFINED, and are typically used for internal communication between the hardware and software components of a VFP implementation (see *Hardware and software implementations* on page C1-5).

C1.2.2 Instructions

Instructions are provided to:

- Load floating-point values into registers from memory, and store floating-point values in registers to memory. Some of these instructions allow multiple register values to be transferred, providing floating-point equivalents to ARM LDM and STM instructions. Among other purposes, such instructions can be used to load and store short vectors of floating-point values.

- Transfer 32-bit values directly between VFP and ARM general-purpose registers.

- Transfer 32-bit values directly between VFP system registers and ARM general-purpose registers.

- Add, subtract, multiply, divide, and take the square root of floating-point register values. These instructions can be used on short vectors as well as on individual floating-point values.

- Copy floating-point values between registers. In the process, the sign bit can be inverted or cleared (or left unchanged), providing negation and absolute value instructions as well as straightforward copies. All of these instructions can also be used on short vectors.

- Perform combined multiply-accumulate operations on floating-point values and short vectors, providing space-efficient equivalents for common sequences of multiply, negate, add, and subtract.

- Perform conversions between single-precision values, double-precision values, unsigned 32-bit integers and two's complement signed 32-bit integers.

• Compare floating-point values in registers with each other or with zero.

C1.2.3 Floating-point exceptions

The VFP architecture supports all five of the floating-point exceptions defined in the IEEE 754 standard:

• Invalid Operation
• Division by Zero
• Overflow
• Underflow
• Inexact.

These are supported in both untrapped and trapped forms:

Untrapped handling of an exception

This causes the appropriate cumulative flag in the FPSCR to be set to 1, and any result registers of the exception-generating instruction to be set to the result values specified by the standard. Execution of the program containing the exception-generating instruction then continues.

Trapped handling of an exception

This is selected by setting the appropriate control bit in the FPSCR. When the exception occurs, a trap handler software routine is called. Details of how trap handler routines are called and of the facilities available to them are IMPLEMENTATION DEFINED.

C1.2.4 Hardware and software implementations

Because of the existence of trapped floating-point exceptions, any implementation of the VFP architecture must include a software component. This is typically installed on the ARM undefined instruction vector, and has the job of catching a trapped exception and converting it into a trap handler call.

The software component of a VFP implementation can perform other tasks in addition to trap handler calls. The division of labour between the hardware and software components of a VFP implementation is IMPLEMENTATION DEFINED.

VFP implementations can be classified according to whether they also include a hardware component:

Software implementation

This implementation consists of software only, with all floating-point arithmetic being emulated by ARM routines. A software implementation is also sometimes called a *VFP emulator*.

Hardware implementation

This implementation contains both hardware and software components. Typically, the hardware is designed to handle all common cases, to optimize performance. When a case where the hardware cannot handle on its own is encountered, the software component (also known as *support code* for the hardware) is called to deal with it. Details of how the hardware and its support code interact are IMPLEMENTATION DEPENDENT.

C1.2.5 Interactions with the ARM architecture

The VFP architecture has been designed to conform fully with the ARM coprocessor architecture. All VFP instructions are special cases of the ARM's generic coprocessor instructions (CDP, LDC, MCR, MRC, and STC), using coprocessor numbers 10 and 11. As a general rule, coprocessor 10 is used for single-precision instructions and coprocessor 11 for double-precision instructions.

All coprocessor 10 and 11 instructions that have not been allocated meanings as VFP instructions are reserved for future expansion of the VFP architecture, and must be treated as UNDEFINED. Hardware coprocessor implementations of the VFP architecture fail to respond to these instructions, causing the ARM's Undefined Instruction exception to occur. For more details, see *Undefined Instruction exception* on page A2-18.

The recommended way for a VFP coprocessor to invoke its support code uses the same mechanism:

1.   Before the VFP hardware is enabled, the support code is installed on the ARM's undefined instruction vector.

2.   When the hardware needs assistance from the support code, it fails to respond to a VFP instruction.

3.   This results in an Undefined Instruction exception, causing the support code to be executed.

In such a system, the support code is responsible for distinguishing these Undefined Instruction exceptions from those caused by the reserved instructions and taking different actions accordingly.

The ARM tests whether a coprocessor instruction satisfies its condition (as described in *The condition field* on page A3-7), using the CPSR flags, and treats it as a NOP if the condition fails. If this happens, the ARM signals coprocessors not to execute the instruction, so they also treat the instruction as a NOP. This implies that all VFP instructions are treated as NOPs if their condition check fails.

The condition code check is based on the ARM processor's CPSR flags, not on the similarly named flags in the VFP FPSCR register. To use the FPSCR flags for conditional execution, they must first be transferred to the CPSR by an FMSTAT instruction.

VFP load and store instructions are allowed to produce Data Aborts, and so VFP implementations are able to cope with a Data Abort on any memory access caused by such instructions.

Interrupts

As described above, hardware VFP implementations typically use the Undefined Instruction exception to communicate between their hardware and software components. Software VFP implementations also use the Undefined Instruction exception, since all coprocessor instructions that are not claimed by a hardware coprocessor are treated as undefined instructions.

Entry to the Undefined Instruction exception causes IRQs to be disabled (see *Undefined Instruction exception* on page A2-18), and they are not normally re-enabled until the exception handler returns. Straightforward use of VFP in a system therefore increases worst case IRQ latency considerably.

It is possible to reduce this IRQ latency penalty considerably by explicitly re-enabling interrupts soon after entry to the Undefined Instruction handler. This requires careful integration of the Undefined Instruction handler into the rest of the operating system. Details of how this should be done are highly system-specific and go beyond the scope of this manual.

In a hardware implementation, if the IRQ handler is going to use the VFP coprocessor itself, there is a second potential cause of increased IRQ latency. This is that a long latency VFP operation initiated by the interrupted program denies the use of the VFP hardware to the IRQ handler for a significant number of cycles.

If a system contains IRQ handlers which require both low interrupt latency and the use of VFP instructions, therefore, it is recommended that the use of the highest latency VFP instructions is avoided. In particular, the use of vector division instructions and vector square root instructions is not recommended in such systems, because these instructions typically have very long latencies.

——— **Note** ———

FIQs are not disabled by entry to the Undefined Instruction handler, and so FIQ latency is not affected by the way that a VFP implementation uses the Undefined Instruction exception.

However, this also means that an FIQ can occur at any point during the execution of a VFP implementation's software component, including during the entry and exit sequences of the Undefined Instruction handler. If a FIQ handler is going to do anything other than leave the VFP implementation's state entirely unchanged, great care must be taken to ensure that it handles every case correctly. This is usually incompatible with the intention that FIQs should provide fast interrupt processing, and so it is recommended that FIQ handlers should not use VFP.

C1.3   Compliance with the IEEE 754 standard

The VFP architecture supplies a subset of IEEE 754 functionality. The following operations are mandatory under the standard, but not supplied by the VFP architecture:

•   the remainder operation

•   the binary ↔ decimal conversions

•   the *Round Floating-Point Number to Integer Value* operation

•   in D variants of the VFP architecture, comparisons directly between single-precision and double-precision values without first converting the single-precision value to double-precision.

To obtain a fully compliant implementation of the standard, the VFP architecture must be augmented with these operations (typically in the form of software library routines).

———— **Note** ————

In some environments, not all of these operations are required. For example, the C language specifies that if a `float` and a `double` are compared, the first argument must be converted to a `double` by the usual binary conversions *before* the comparison is performed. So, C code never specifies a direct comparison of a single-precision value and a double-precision value.

Also, when the *Flush to Zero* (FZ) bit in the FPSCR is set to 1, the way the VFP architecture handles denormalized numbers and underflow exceptions does not comply with the standard. To obtain fully compliant behavior from the VFP architecture, the FZ bit must be set to 0 (see *Flush-to-zero mode* on page C2-13 for more details).

C1.4 IEEE 754 implementation choices

Many design choices about a compliant floating-point system are left as an implementation option by the IEEE 754 standard. The VFP architecture specifies how many of these choices are to be made. The rest of this section briefly describes these implementation choices.

C1.4.1 Supported formats

The VFP architecture supports the *basic single* floating-point format from the standard, and D variants also support the *basic double* floating-point format. These are known as *single-precision* and *double-precision* in this manual.

The standard's extended formats are not supported.

Supported integer formats are unsigned 32-bit integers and two's complement signed 32-bit integers.

C1.4.2 NaNs

The IEEE 754 standard only specifies that there must be at least one signaling NaN and at least one quiet NaN, and partly specifies what the representation of NaNs should be (for any NaN, the exponent field should be maximum, and the fraction field non-zero). The VFP architecture specifies its NaNs more fully:

- In each format, *all* values with the exponent field maximum and the fraction field non-zero are valid NaNs. Two such values represent distinct NaNs if their sign bits and/or fraction fields are different.

- Copying a signaling NaN with a change of format does not generate an Invalid Operation exception.

- Signaling NaNs are distinguished from quiet NaNs by the most significant fraction bit. The NaN is signaling if this bit is 0, and quiet if it is 1.

- There are precise rules in the VFP architecture about which NaN is produced for each operation with a NaN result. These rules are described in *NaNs* on page C2-5.

———— **Note** ————

The fact that NaNs whose sign or fraction bits differ are treated as distinct NaNs in the VFP architecture does not mean that the floating-point comparison instructions can be used to distinguish them from each other. The IEEE 754 standard requires all NaNs to compare as *unordered* with every value, including themselves.

What it does mean is that the distinct NaNs can be distinguished by using ARM code that looks at their precise bit patterns, and that the NaN handling rules are designed not to change bits in NaN values except where this is required by the standard.

C1.4.3 Comparison results

The results of comparison instructions are delivered as condition codes. In particular, they are flag combinations (N, Z, C, and V), compatible with those used by the ARM program status registers.

To assist with the alternative approach of testing predicates, each comparison instruction is supplied in two variants whose behavior differs with respect to NaNs, and the flag combinations (N, Z, C, and V) for the four possible comparison results are chosen to maximize the number of predicates that can be tested with a single ARM condition check. See *Testing the IEEE 754 predicates* on page C3-8 for more details.

C1.4.4 Underflow exception

Underflow is detected using the *after rounding* form of tininess and the *denormalization loss* form of loss of accuracy, as defined in the IEEE 754 standard.

C1.4.5 Exception traps

The FPSCR contains bits to specify whether exception traps are enabled, and the VFP implementation determines whether a trapped exception as defined by the IEEE 754 standard does in fact occur. All further details of trapped exception handling are IMPLEMENTATION DEFINED.

Chapter C2
VFP Programmer's Model

This chapter gives details of the VFP programmer's model. It contains the following sections:

- *Floating-point formats* on page C2-2
- *Rounding* on page C2-9
- *Floating-point exceptions* on page C2-10
- *Flush-to-zero mode* on page C2-13
- *Default NaN mode* on page C2-14
- *RunFast mode* on page C2-15
- *Floating-point general-purpose registers* on page C2-16
- *System registers* on page C2-21
- *Reset behavior and initialization* on page C2-28.

C2.1 Floating-point formats

This section outlines the basic single-precision and double-precision floating-point formats, as defined by the IEEE 754 standard and used by the VFP architecture. In addition, it describes VFP-specific details of these formats that are left open by the standard.

All versions and variants of the VFP architecture support the single-precision format. D variants also support the double-precision format. The VFP architecture does not support either of the extended formats described in the IEEE 754 standard.

This section is only intended as an introduction to these formats and to the various types of value they can contain, not as comprehensive reference material on them. For full details, especially of the handling of infinities, NaNs and signed zeros, see the IEEE 754 standard.

C2.1.1 Single-precision format

A single-precision value is a 32-bit word, and must be word-aligned when held in memory. It has the following format:

31	30	23	22	0
S	exponent		fraction	

The value represented depends primarily on the exponent field:

- If 0 < exponent < 0xFF, the value is a *normalized number* and is equal to:

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

  The *mantissa* of the value is the number 1.fraction, consisting of:

  — 1

  — a binary point

  — the 23 fraction bits.

  The mantissa therefore lies in the range $1 \leq \text{mantissa} < 2$ and is a multiple of 2^{-23}.

  The *unbiased exponent* of the value is the power to which 2 is raised in this formula. In this case, it is (exponent−127).

  The minimum positive normalized number is 2^{-126}, or approximately 1.175×10^{-38}. The maximum positive normalized number is $(2-2^{-23}) \times 2^{127}$, or approximately 3.403×10^{38}.

- If exponent == 0, the value is either a zero or a *denormalized number*, depending on the fraction bits:

  — If fraction == 0, the value is a zero.

    There are two distinct zeros:

    +0   with S==0

    −0   with S==1.

These behave identically in most circumstances, including getting an *equal* result if +0 and −0 are compared as floating-point numbers. However, they yield different results in some exceptional circumstances (for example, they affect the sign of the infinity produced as the default result for a Division by Zero exception). They can also be distinguished from each other by performing an integer comparison of the two words.

— If fraction != 0, the value is a denormalized number and is equal to:

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

In this case, the mantissa of the value has a zero before the binary point, rather than the one used by a normalized number. It lies in the range $0 < \text{mantissa} < 1$ and is a multiple of 2^{-23}. The value's unbiased exponent is −126.

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

• If exponent == 0xFF, the value is either an *infinity* or a *Not a Number* (NaN), depending on the fraction bits.

If fraction == 0, the value is an infinity. There are two infinities:

+∞ Has S==0 and represents all positive numbers which are too big to be represented accurately as a normalized number.

−∞ Has S==1 and represents all negative numbers which are too big to be represented accurately as a normalized number.

If fraction != 0, the value is a NaN, and can be either a *quiet NaN* or a *signaling NaN* (see *NaNs* on page C2-5 for details of these types of NaN).

In the VFP architecture, the two types of NaN are distinguished on the basis of their most significant fraction bit (bit[22]):

— If bit[22] == 0, the NaN is a signaling NaN. The sign bit can take any value, and the remaining fraction bits can take any value except all zeros, so there are $2 \times (2^{22}-1) = 8388606$ possible signaling NaNs.

— If bit[22] == 1, the NaN is a quiet NaN. The sign bit and remaining fraction bits can take any value, so there are $2 \times 2^{22} = 8388608$ possible quiet NaNs.

Two NaNs are treated as being different values in the VFP architecture if their sign bits and/or any of their fraction bits differ. This implies that all 2^{32} possible word values are treated as distinct from each other by the VFP architecture.

——— **Note** ———

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

However, different NaNs can be distinguished by using integer comparisons. Also, the rules for handling NaNs are designed not to arbitrarily change one NaN into another (see *NaNs* on page C2-5).

These rules about NaNs also ensure that single-precision registers can be used to hold integer values without any risk of corrupting them (see *Holding integers in single-precision registers* on page C2-20).

C2.1.2 Double-precision format

A double-precision value consists of two 32-bit words, with the following formats:

Most significant word:

31	30	20	19	0

S	exponent	fraction[51:32]

Least significant word:

31	0

fraction[31:0]
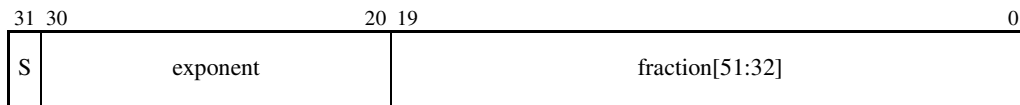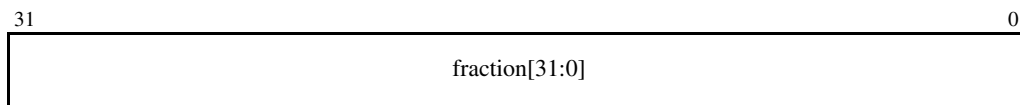
When held in memory, the two words must appear consecutively and must both be word-aligned. The order of the two words depends on the endianness of the memory system:

- In a little-endian memory system, the least significant word appears at the lower memory address and the most significant word at the higher memory address.

- In a big-endian memory system, the most significant word appears at the lower memory address and the least significant word at the lower memory address.

A VFP implementation must use the same endianness as the ARM implementation it is attached to. If the ARM implementation has configurable endianness, double-precision values must not be loaded or stored before the ARM processor endianness has been set to match that of the memory system (see *Endian Support* on page A2-29 for more details).

——— **Note** ———

The word order defined here for the VFP architecture differs from that of the earlier FPA floating-point architecture. In the FPA architecture, the most significant word always appeared at the lower memory address, with the least significant word at the higher, regardless of the memory system endianness.

Double-precision values represent numbers, infinities and NaNs analogously to single-precision values:

- If $0 < \text{exponent} < \texttt{0x7FF}$, the value is a normalized number and is equal to:

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

The mantissa of the value is the number 1.fraction, consisting of a one, followed by a binary point, followed by the 52 fraction bits. The mantissa therefore lies in the range $1 \leq \text{mantissa} < 2$ and is a multiple of 2^{-52}.

The unbiased exponent of the value is (exponent-1023).

The minimum positive normalized number is 2^{-1022}, or approximately 2.225×10^{-308}. The maximum positive normalized number is $(2-2^{-52}) \times 2^{1023}$, or approximately 1.798×10^{308}.

- If exponent == 0, the value is either a zero or a denormalized number, depending on the fraction bits.

  If fraction == 0, the value is a zero. There are two distinct zeros which behave analogously to the two single-precision zeros:

  +0   with S==0

  −0   with S==1.

  If fraction != 0, the value is a denormalized number and is equal to:

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

  In this case, the mantissa of the value has a zero before the binary point, rather than the one used by a normalized number. It lies in the range 0 < mantissa < 1 and is a multiple of 2^{-52}. The unbiased exponent of the value is −1022.

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

- If exponent == 0x7FF, the value is either an infinity or a NaN, depending on the fraction bits.

  — If fraction == 0, the value is an infinity. As for single-precision, there are two infinities:

    +∞   Plus infinity with S==0

    +∞   Minus infinity with S==1.

  — If fraction != 0, the value is a NaN. The rules about interpreting NaN values are analogous to those for single-precision TBD repeat rather than xreffing

    The type of a NaN is determined by the most significant fraction bit (bit[19] of the most significant word). A 0 indicates a signaling NaN and a 1 indicates a quiet NaN.

    Two NaNs whose sign bits and/or fractions differ are different NaNs.

C2.1.3   NaNs

NaNs are special floating-point values which can be used when neither a numeric value nor an infinity is appropriate. There are two types of NaN, each of which can be used for a variety of purposes:

Quiet NaNs   These propagate unchanged through most floating-point operations. They can be generated by floating-point arithmetic operations in some rare circumstances when there is no other sensible result. Any further calculations which depend on the result of such an operation then also produce a quiet NaN result. (Quiet NaNs can only be generated in this way if the associated Invalid Operation exception is untrapped. If it is trapped, a trap handler is called instead.) Another typical use for quiet NaNs is to represent missing or unavailable data values. The results of any calculations that depend on the missing values are then also quiet NaNs.

Signaling NaNs   These cause an Invalid Operation exception whenever any floating-point operation receives a signaling NaN as an operand.

                        One possible use for signaling NaNs is in debugging, to track down some uses of uninitialized variables. To do this, pre-load memory with copies of a signaling NaN, then load and run the program with Invalid Operation traps enabled. Any floating-point operation whose operand has been loaded from uninitialized memory then calls the Invalid Operation trap handler.

The IEEE 754 standard does not specify how the two types of NaN are distinguished or how many different NaNs of each type can exist in a floating-point system. However, these details are specified by the VFP architecture, as described in *Single-precision format* on page C2-2 and *Double-precision format* on page C2-4.

The following subsections describe the main requirements of the IEEE 754 standard about how floating-point operations involving NaNs behave, and additional requirements on such operations imposed by the VFP architecture.

Instructions with non floating-point results

The VFP architecture contains instructions to convert floating-point values to integers. In accordance with the IEEE 754 standard, these instructions always generate an Invalid Operation exception if their operand is a NaN, regardless of whether it is a signaling NaN or a quiet NaN. If this exception is untrapped, the VFP architecture specifies that the integer result must be 0.

The VFP architecture also contains comparison instructions, which deliver condition code results. These instructions generate Invalid Operation exceptions for signaling NaN operands. For quiet NaN operands, some of them also generate Invalid Operation exceptions, while others generate an *unordered* condition code result. The condition code result is also *unordered* in all cases where an Invalid Operation exception is generated but the exception is untrapped. For more details, see *Comparison instructions* on page C3-6.

All other VFP instructions that process NaNs have floating-point result values.

Instructions with floating-point results

If one or more of the operands to an operation with a floating-point result is a NaN, the IEEE 754 standard requires that:

*   If any of the NaN operands is a signaling NaN, an Invalid Operation exception must be generated. If this exception is untrapped, the result must be a quiet NaN.

*   If all of the NaN operands are quiet NaNs, the result must be a quiet NaN, and must be equal to one of the NaN operands.

——— **Note** ———

For this purpose, the standard permits some copy operations on floating-point numbers to be treated as non floating-point operations, so that they do not process NaNs in this fashion. The VFP architecture requires these copy operations to be treated as non floating-point operations.

Instructions affected by this are described in *Copy, negation and absolute value instructions* on page C3-11, *Load and Store instructions* on page C3-13 and *Single register transfer instructions* on page C3-17.

Most floating-point instructions in the VFP architecture use the same format for their operands and results. For these, the VFP architecture specifies that the correct quiet NaN result in either of the above cases is determined as follows:

1. For instructions acting on vector operands, rules 2 through 4 below are applied independently to each individual operation on vector elements.

2. The FMAC, FMSC, FNMAC, and FNMSC instructions each specify two floating-point operations, each with two operands. If either of the operands to the first operation is a NaN, its result is determined according to rules 3 and 4 below. Then the third operand and result of the first operation (with its sign bit inverted for FNMAC and FNMSC) become the operands of the second operation. If either of them is a NaN, the final result is determined according to rules 3 and 4 below.

3. If an operand is a signaling NaN, the result is the quiet NaN constructed by taking a copy of that operand and changing its most significant fraction bit from 0 to 1. If both operands of a two-operand operation are signaling NaNs, the first operand is the one used to generate the result in this fashion.

4. If no operand is a signaling NaN, but an operand is a quiet NaN, the result is a copy of the quiet NaN operand. If both operands of a two-operand operation are quiet NaNs, the first operand is the one copied to generate the result.

The IEEE 754 standard also specifies that an Invalid Operation exception must be generated for certain operations whose operands are not NaNs. The following operations yielding floating-point results can cause this to happen:

*   Additions, when the two operands are infinities with opposite signs. VFP instructions affected by this are FADD, FMAC, and FNMAC.

*   Subtractions, when the two operands are infinities with the same sign. VFP instructions affected by this are FMSC, FNMSC, and FSUB.

*   Multiplications, when one operand is a zero and the other is an infinity. VFP instructions affected by this are FMAC, FMSC, FMUL, FNMAC, FNMSC, and FNMUL.

*   Divisions, when both operands are zeros or both operands are infinities. The only VFP instruction affected by this is FDIV.

*   Square roots, whose operands are negative, including $-\infty$ (minus infinity) but excluding -0. The only VFP instruction affected by this is FSQRT.

In each case, if the exception is untrapped, the result must be a quiet NaN. The VFP architecture specifies that the quiet NaN produced in these cases must have sign bit equal to 0, most significant fraction bit equal to 1, and all remaining fraction bits equal to 0.

Special cases

There are two instructions whose operands and results have different floating-point formats. These have special rules for handling NaNs, as follows:

• The FCVTDS instruction converts a single-precision value to double-precision. If its operand is a single-precision quiet NaN, the result is the double-precision quiet NaN with:

```
S               = S bit of operand
fraction[51:29] = fraction[22:0] of operand
fraction[28:0]  = 0
```

If its operand is a single-precision signaling NaN, an Invalid Operation exception is generated. If the exception is untrapped, the result is the double-precision quiet NaN with:

```
S               = S bit of operand
fraction[51]    = 1
fraction[50:29] = fraction[21:0] of operand
fraction[28:0]  = 0
```

• The FCVTSD instruction converts a double-precision value to single-precision. If its operand is a double-precision quiet NaN, the result is the single-precision quiet NaN with:

```
S               = S bit of operand
fraction[22:0]  = fraction[51:29] of operand
```

If its operand is a double-precision signaling NaN, an Invalid Operation exception is generated. If the exception is untrapped, the result is the single-precision quiet NaN with:

```
S               = S bit of operand
fraction[22]    = 1
fraction[21:0]  = fraction[50:29] of operand
```

C2.2   Rounding

Floating-point arithmetic inherently only has limited accuracy, because the exact mathematical result of an arithmetic operation often has more significant bits than can fit in its destination format. To deal with this, the result is *rounded* to fit in the destination format, by choosing a representable number in that format which is a close approximation to the exact result.

The IEEE 754 standard specifies four *rounding modes*, each of which specifies how the exact result of an operation is rounded. In the following descriptions of the rounding modes, the *rounding error* is defined to be the value of:

(rounded result) - (exact result)

The rounding modes are as follows:

Round to Nearest (RN) mode

In this mode, the rounded result is the nearest representable number to the unrounded result, that is, the representable number that minimizes abs(rounding error). If the unrounded result lies precisely halfway between two representable numbers, the one whose least significant bit is 0 is used.

This is the default rounding mode, and generally yields the most accurate results. The other rounding modes are mostly used for specialized purposes, such as interval arithmetic.

Round towards Plus Infinity (RP) mode

In this mode, the rounded result is the nearest representable number which is greater than or equal to the exact result, that is, the one that minimizes abs(rounding error) subject to the requirement (rounding error) ≥ 0. If the exact result is greater than the largest positive normalized number of the destination format, the rounded result is $+\infty$ (plus infinity).

Round towards Minus Infinity (RM) mode

In this mode, the rounded result is the nearest representable number which is less than or equal to the exact result, that is, the one that minimizes abs(rounding error) subject to the requirement (rounding error) ≤ 0. If the exact result is less than the largest negative normalized number of the destination format, the rounded result is $-\infty$ (minus infinity).

Round towards Zero (RZ) mode

In this mode, results are rounded to the nearest representable number which is no greater in magnitude than the unrounded result, that is, the one that minimizes abs(rounding error) subject to the requirement abs(rounded result) $\leq$ abs(exact result).

C2.3 Floating-point exceptions

The IEEE 754 standard specifies five classes of floating-point exception:

Invalid Operation exceptions

Occur in various cases where neither a numeric value nor an infinity is a sensible result of a floating-point operation, and also when an operand of a floating-point operation is a signaling NaN. For more details of Invalid Operation exceptions, see *NaNs* on page C2-5.

Division by Zero exceptions

Occur when a normalized or denormalized number is divided by a zero.

Overflow exceptions

Occur when result of a floating-point operation is too big in magnitude for it to be represented in the destination format without an unusually large rounding error for the rounding mode in use. More precisely, the *ideal rounded result* of a floating-point operation is defined to be the result that its rounding mode would produce if the destination format had no limits on the unbiased exponent range. If the ideal rounded result has unbiased exponent too big for the destination format (that is, >127 for single-precision or >1023 for double-precision), it differs from the actual rounded result, and an Overflow exception occurs.

Underflow exceptions

tbd rewrite to account for changed functionality Occur when a floating-point operation's result is too small in magnitude for it to be represented in the destination format without an unusually large rounding error for the rounding mode in use. The exact conditions for an Underflow exception to occur vary slightly depending on the value of the *Underflow exception enable* (UFE) bit (which is bit[11] of the FPSCR), that is, on whether the Underflow exception is to be trapped.

In each case, one condition is that the ideal rounded result (as defined above) has an unbiased exponent too small for the destination format (that is, < -126 for single-precision or < -1022 for double-precision). If UFE == 1, this is the only condition and a trapped Underflow exception occurs when it is detected. If UFE == 0, an untrapped Underflow exception only occurs if the ideal rounded result is also different from the result produced by actual rounding (which might be a zero, a denormalized number or the minimum possible normalized number).

—— **Note** ——

The IEEE 754 standard leaves two choices open in its definition of the underflow exception. In the terminology of the standard, the above description means that the VFP architecture requires these choices to be:

- the *after rounding* form of *tininess*
- the *denormalization loss* form of *loss of accuracy*.

Inexact exceptions

tbd rewrite to account for changed functionality Occur whenever the rounded result of a floating-point operation is different from its exact unrounded result, that is, whenever a non-zero rounding error occurs. They also occur:

- whenever an untrapped Overflow exception occurs
- whenever a denormalized operand is treated as +0 or a result is forced to +0 by flush-to-zero mode.

——— **Note** ———

Despite being called an exception, the Inexact exception occurs frequently in normal floating-point calculations. Other than in some specialized uses of floating-point arithmetic, the Inexact exception does not indicate that anything unusual has occurred.

Each of these exceptions can be handled in one of two ways, selected by a *trap enable bit* associated with the exception:

Trap enable bit is 0

Untrapped exception handling is selected.

This causes the result of the operation to be a default value specified by the IEEE 754 standard, and a *cumulative exception bit* associated with the exception becomes 1. Table C2-1 on page C2-12 shows how the result value is determined for each exception.

The cumulative exception bits can only become 0 as the result of an explicit write to the FPSCR using the FMXR instruction. Other floating-point instructions only leave them unchanged (if no untrapped exceptions occurred) or set one or more of them to 1 depending on which untrapped exceptions occurred. A program can therefore test whether untrapped exceptions occurred during a calculation, by setting these bits to zero before the calculation and testing them afterwards.

Trap enable bit is 1

Trapped exception handling is selected.

This causes a trap handler routine for the exception to be called. Details of how trap handlers are selected and of the interfaces via which they are called are IMPLEMENTATION DEFINED.

The call to the trap handler routine is allowed to be *imprecise*, that is, it might occur at a later point during program execution than the floating-point instruction that caused the exception. However, it always occurs before execution of any subsequent instruction that depends on the results of that instruction, or of any *serializing instruction* (see *FMRX* on page C4-63 and *FMXR* on page C4-78).

Trapped exception handling does not cause the cumulative exception bit to become set. If this behavior is desired, the trap handler routine can use an FMRX/ORR/FMXR sequence on the FPSCR to set the bit.

In addition to the IEEE 754 exceptions, VFPv2 provides an *Input Denormal Condition* (IDC). This condition is handled in exactly the same way as the exceptions.

Copyright © 1996–2003 ARM Limited. All rights reserved.
Confidential - Draft

Table C2-1 Exception default results

Exception type	Default result for positive sign		Default result for negative sign	
Invalid Operation	Quiet NaN		Quiet NaN	
Division by Zero	$+\infty$ (plus infinity)		$-\infty$ (minus infinity)	
Overflow	RN,RP:	$+\infty$ (plus infinity)	RN,RM:	$-\infty$ (minus infinity)
	RM,RZ:	+MaxNorm	RP,RZ:	−MaxNorm
Underflow	Normal rounded result		Normal rounded result	
Inexact	Normal rounded result		Normal rounded result	

The following notes apply to Table C2-1:

- For Invalid Operation exceptions, see *NaNs* on page C2-5 for details of which quiet NaN is produced as the default result.

- For Division by Zero exceptions, the default result depends on the sign bit as normally determined for a division - that is, on the exclusive OR of the two operand sign bits.

- For Overflow exceptions, the default result depends on the sign bit as normally determined for the overflowing operation, and also on which rounding mode is being used. MaxNorm means the maximum normalized number of the destination precision.

C2.3.1 Combinations of exceptions

It is possible for more than one exception to occur on the same operation. The only combinations of exceptions that can occur are Overflow/Inexact and Underflow/Inexact. In these cases, the Inexact exception is treated as lower priority, as follows:

- If the Overflow or Underflow exception is trapped, its trap handler is called. It is IMPLEMENTATION DEFINED whether the parameters to the trap handler include information about the Inexact exception. Apart from this, the Inexact exception is ignored in this case.

- If the Overflow or Underflow exception is untrapped, its cumulative bit is set to 1 and its default result is evaluated. Then the Inexact exception is handled normally, with this default result being treated as the normal rounded result of the operation.

C2.4 Flush-to-zero mode

The performance of some VFP implementations is significantly lower than normal when performing calculations involving denormalized numbers and Underflow exceptions. Typically, this occurs for hardware implementations which only handle normalized numbers and zeros in hardware, and invoke their support code when they encounter other types of value.

If a significant number of the operands and intermediate results in an algorithm are denormalized numbers, this can result in a considerable loss of performance. In some (but not all) of these algorithms, this performance can be recovered by replacing the denormalized operands and intermediate results with zeros, without significantly affecting the accuracy of their final results. To allow this optimization, VFP implementations have a special processing mode called *flush-to-zero mode*. The behavior in flush-to-zero mode differs from normal IEEE 754 arithmetic in the following ways:

- All inputs to floating-point operations which are denormalized numbers are treated as though they were +0.

- All results of single-precision floating-point operations which lie in the range $-2^{-126} < x < +2^{-126}$ before rounding are forced to +0.

- All results of double-precision floating-point operations which lie in the range $-2^{-1022} < x < +2^{-1022}$ before rounding are forced to +0.

- Whenever an operation has an operand treated as +0 or a result forced to +0 in any of the above ways, an Inexact exception occurs (this normally just causes the IXC bit in the FPSCR to become set).

- Underflow exceptions never occur in flush-to-zero mode.

Copy operations are not treated as floating-point operations for the purpose of flush-to-zero mode. The operations not affected by flush-to-zero mode are precisely the same as those that do not generate Invalid Operation exceptions when their operands are signaling NaNs. For more details, see *Copy, negation and absolute value instructions* on page C3-11, *Load and Store instructions* on page C3-13, and *Single register transfer instructions* on page C3-17.

——— **Note** ———

Flush-to-zero mode is incompatible with the IEEE 754 standard, and must not be used when IEEE 754 compatibility is a requirement. Flush-to-zero mode must be treated with great care. As stated above, it can lead to a major performance increase on some algorithms, but there are a number of pitfalls when using it. These pitfalls include:

- On many algorithms, it has no noticeable effect, because the algorithm does not normally use denormalized numbers.

- On many other algorithms, it can cause exceptions to occur or seriously impact the accuracy of the results of the algorithm.

It is strongly recommended that flush-to-zero mode is only used after ascertaining that performance is indeed being degraded by the occurrence of denormalized numbers, and that its use is not going to upset the accuracy and/or stability of the algorithm.

C2.5 Default NaN mode

VFPv2 introduces *Default NaN mode* (DN mode).

DN mode is selected by setting the DN bit, bit[25], in the FPSCR. The default is 0, disabled. When set, this bit specifies a behavior that is consistent with the IEEE 754 but different from contemporary general-purpose or embedded practice.

The IEEE 754 specifies that an operation involving a NaN returns a QNaN. In most contemporary floating-point implementations the fraction bits returned are the fraction bits of the input NaN, or one of the input NaNs if there are more than one. Which input NaN is returned is specified in the architecture. This is the VFPv2 behavior when Default NaN mode is not enabled.

In DN mode, any arithmetic operation involving one or more input NaNs, quiet or signalling, or an invalid result that returns a NaN, returns the default NaN. The format of the default NaN is shown in Table C2-2.

The non-arithmetic operations FCPY, FABS, and FNEG process NaNs without altering the fraction bits. No exception status bits are set for these instructions when a NaN is involved.

C2.5.1 IOE and IOC FPSCR Bits

The functionality of the IOC bit is not affected by DN mode. It is governed by the specification of the Invalid exception in the IEEE 754 specification regarding NaN inputs.

In DN mode the presence of an input NaN causes a bounce to Support Code only when one or more of the input NaNs are signalling and the IOE trap enable bit is set.

C2.5.2 Format of the Default NaN

The default NaN for ARM floating point processors is shown in Table C2-2.

Table C2-2 Default NaN encoding

	single-precision	double-precision
Sign	0	0
Exponent	0xFF	0x7FF
Fraction	Bit[22] == 1	Bit[51] == 1
	Bits[21:0] == 0	Bits[50:0] == 0

C2.6   RunFast mode

TBD Not sure what to say here. Surely we don't want to give lots of VFP10 details - just some general information about the principle?

When the DN and FZ bits in the FPSCR are enabled, and the IXE enable bit is not set, the VFP10 is described as operating in RunFast mode. In RunFast mode no bouncing is possible except when user trap enable bits are set. When the VFP is in RunFast mode and no enable bits are set, instructions which would be required to stal,l to preserve source registers of a preceding instruction, are allowed to proceed without stalling. For example, in the following code sequence the FLDS is overwriting one of the source registers of the FMACS. To guarantee the FMACS source registers are preserved for a bounce condition, the FLDS stalls until the status of the FMACS is known. In the VFP10 this is a one-cycle stall. In RunFast mode, with no exception enable bits set, this stall condition is not required.

Examples:

```
FMACS s4, s5, s6
FLDS s5, [r4]
```

Copyright © 1996–2003 ARM Limited. All rights reserved.
Confidential - Draft

C2.7 Floating-point general-purpose registers

A VFP implementation contains 32 general-purpose registers, each capable of holding a single-precision floating-point number or a 32-bit integer. These are named S0-S31.

In D variants of the VFP architecture, these registers are also treated as 16 double-precision registers, with names D0-D15. Double-precision register D0 overlaps single-precision registers S0 and S1, double-precision register D1 overlaps single-precision registers S2 and S3, and so on, as shown in Figure C2-1.

S0	S1		D0
S2	S3		D1
S4	S5		D2
S6	S7		D3
S8	S9		D4
S10	S11		D5
S12	S13		D6
S14	S15		D7
S16	S17	overlapped with	D8
S18	S19		D9
S20	S21		D10
S22	S23		D11
S24	S25		D12
S26	S27		D13
S28	S29		D14
S30	S31		D15
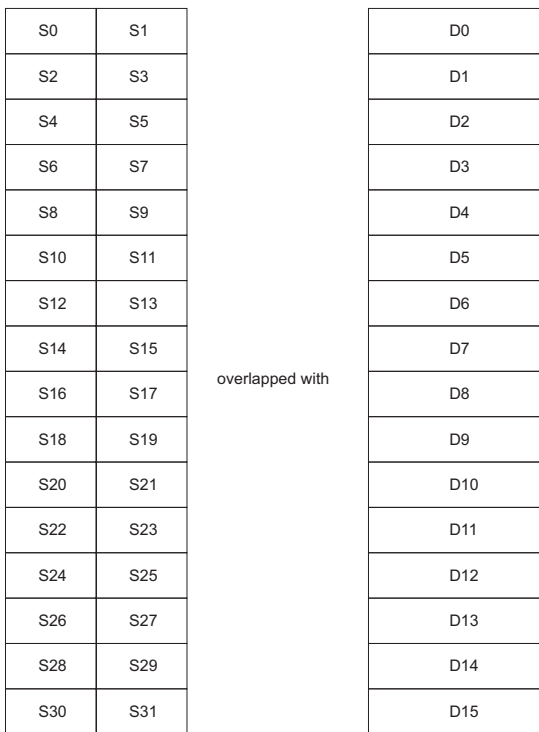
Figure C2-1 VFP general-purpose registers

The VFP architecture does not specify how a double-precision register is overlapped with its pair of single-precision registers. Different implementations make different decisions about which bits are overlapped and about how the overlapping bits match up.

Also, implementations are free to convert numbers of either precision (or both) to an internal register format when they are loaded (or transferred from the ARM), and convert them back when stored (or transferred to the ARM), as long as the overall results are correct.

As a result, software must not rely on any particular type of overlap between the registers. More formally, the rules for this are:

• When a single-precision value or a 32-bit integer is written to single-precision register Si, the value in the overlapped double-precision register D(i >> 1) becomes UNPREDICTABLE.

• When a double-precision value is written to double-precision register Di, the values in the overlapped single-precision registers $S(2 \times i)$ and $S(2 \times i + 1)$ become UNPREDICTABLE.

• Any instruction which reads one or more registers with UNPREDICTABLE contents produces UNPREDICTABLE results.

Accordingly, most VFP instructions must only be used when it is known that the values in their source registers have the precision expected by the instruction. No facility exists in the architecture to detect or correct an inconsistency between the precision expected by an instruction and the precision of the numbers last written to its source registers. Such inconsistencies produce UNPREDICTABLE behavior.

C2.7.1   Storing and reloading values of unknown precision

Programs sometimes need to store register values to memory and later reload them without determining whether they contain single-precision or double-precision values. Two typical cases in which this happens are:

• Procedure-calling standards often specify that registers are *callee-save registers* (that is, that the called procedure must preserve them). If the called procedure needs to use a callee-save register, its entry sequence must store the register value to the stack. Later, the return sequence of the procedure must reload the value from the stack to restore the original contents of the register.

  However, the contents of the register(s) being stored on the stack depend on how they were being used by the caller, and different callers can use the registers differently. So the entry sequence of the called procedure must treat the callee-save registers as containing values of unknown precision.

• Process swap code needs to store the contents of registers when a process is swapped out, and later reload them when the process is swapped back in. As different processes probably use the registers in different ways, process swap code needs to treat the VFP registers as containing values of unknown precision.

Two VFP instructions (FLDMX and FSTMX) are used in such situations. These instructions are exceptions to the normal rule that the source precision must match the precision of the instruction.

FSTMX   Stores one or more double-precision registers, using an IMPLEMENTATION DEFINED memory format

FLDMX   Reloads registers that have been stored in that format.

The only architectural constraints on the FSTMX/FLDMX format are that:

• N double-precision registers are stored in at most 2N+1 words.

Copyright © 1996–2003 ARM Limited. All rights reserved.
Confidential - Draft

- A matching `FLDMX` reloads the original contents of the registers correctly, regardless of whether they originally contained single-precision or double-precision values. For this purpose, a *matching FLDMX* means one that loads precisely the same set of registers as the `FSTMX` stored and generates the same memory addresses as the `FSTMX`.

The only operation which is normally performed on data stored with `FSTMX` is to reload it using a matching `FLDMX`. However, debug software might need to interpret and/or modify the contents of stack frames or process control blocks, and so might need to know the `FSTMX`/`FLDMX` memory format. There are two standard options for the format:

- In standard format 1, `FSTMX` stores the register contents in exactly the same way as an `FSTMD` of the same registers would store them. The first double-precision register value is stored as two words, in the correct order for the configured endianness of the processor. Then the next double-precision register value is stored similarly, and so on, until the N registers have been stored in 2N memory words. The (2N+1)th memory word is unused.

  The matching `FLDMX` instruction reloads the data as double-precision values, precisely like an `FLDMD` of the same registers. This format is only used for implementations in which storing and reloading double-precision registers in this fashion correctly reload their contents even if they happen to contain single-precision values.

- In standard format 2, the first word stored is a *format word*, and then each double-precision register specified in the instruction is stored as two words, in ascending order of register number. This option uses all 2N+1 available words of memory.

  The format word has the following format:

31	16 15	0
SBZ		format bits
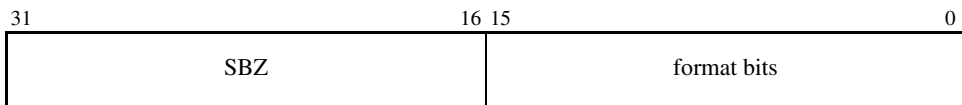
  For each double-precision register Di which is stored by the `FSTMX` instruction, bit[i] of the format word indicates how it is stored:

  — If bit[i] == 0, the register contained single-precision data, and the two words stored are the contents of $S(2 \times i)$ followed by the contents of $S(2 \times i + 1)$, both in single-precision format.

  — If bit[i] == 1, the register contained double-precision data, and the two words stored are the contents of Di, stored as two words in the correct order for the configured endianness of the processor.

  For each double-precision register Di which is not stored by the `FSTMX` instruction, bit[i] of the format word is UNPREDICTABLE.

Example

For example, Figure C2-2 on page C2-19 shows how the instruction:

```
FSTMIAX Rn,{D4-D6}
```

would store the registers using each standard option, on the assumption that D4 and D6 contain double-precision values and S10 and S11 (which overlap D5) contain single-precision values.

Address	Standard format 1	Standard format 2
Rn+24	Unused	D6, second word
Rn+20	D6, second word	D6, first word
Rn+16	D6, first word	S11
Rn+12	D5, second word	S10
Rn+8	D5, first word	D4, second word
Rn+4	D4, second word	D4, first word
Rn	D4, first word	Format word

Figure C2-2 Standard FSTMX/FLDMX memory formats

In the second column of Figure C2-2, the format word is:

31	16	15	7	6	4	3	0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0		UNP		1 0 1		UNP	

C2.7.2 Short vectors

The single-precision registers can be used to hold short vectors of up to 8 single-precision values. Arithmetic operations on all the elements of such a vector can be specified by just one single-precision arithmetic instruction. For details of how this is done, see *Addressing Mode 1 - Single-precision vectors (non-monadic)* on page C5-2 and *Addressing Mode 3 - Single-precision vectors (monadic)* on page C5-14.

Similarly, the double-precision registers can be used to hold short vectors of up to 4 double-precision values, and double-precision arithmetic instructions can specify operations on these vectors. For details, see *Addressing Mode 2 - Double-precision vectors (non-monadic)* on page C5-8 and *Addressing Mode 4 - Double-precision vectors (monadic)* on page C5-19.

C2.7.3   Holding integers in single-precision registers

Each single-precision register can hold a 32-bit integer instead of a single-precision floating-point number. The register contents are identical for a 32-bit integer and for the single-precision value represented by the same word. This means that `FMRS`, `FMSR` and the single-precision load/store instructions can be used to transfer either integers or single-precision values.

The single-precision floating-point number represented by the same word as a 32-bit integer does not normally have the same value as the integer. For example, the integers 2 and −1 are represented by the words `0x00000002` and `0xFFFFFFFF`. As single-precision floating-point numbers, the same words represent the denormalized number 2^{-148} and a quiet NaN respectively.

An integer held in a floating-point register can therefore not be used directly as a single-precision value, nor can a single-precision value be used directly as an integer. If conversions between integers and floating-point values are wanted, explicit conversion instructions must be used, as described in the following two subsections.

Floating-point to integer

Two instructions are used to convert a floating-point number to an integer:

1. The first instruction is `FTOSID`, `FTOSIS`, `FTOUID`, or `FTOUIS`, depending on whether the floating-point operand is double-precision or single-precision, and whether a signed or unsigned integer result is wanted. After this instruction, the required result is held as an integer in a single-precision register.

    The special forms `FTOSIZD`, `FTOSIZS`, `FTOUIZD`, and `FTOUIZS` of these instructions allow the conversion to be done using Round towards Zero (RZ) mode, without changing the rounding mode specified by the FPSCR. This is the form of floating-point to integer conversion required by the C, C++ and related languages.

2. The second instruction is typically an `FMRS` instruction, which transfers the integer result to an ARM register, but can also be various other instructions (see *Conversion instructions* on page C3-10).

Integer to floating-point

Similarly, two instructions are used to convert an integer to a floating-point number:

1. The first instruction is typically `FMSR`, to transfer the integer operand to a single-precision register, but can also be various other instructions (see *Conversion instructions* on page C3-10).

2. The second instruction is `FSITOD`, `FSITOS`, `FUITOD` or `FUITOS`, depending on whether the integer operand is to be treated as signed or unsigned and whether a double-precision or single-precision floating-point result is wanted.

C2.8 System registers

A VFP implementation contains three or more special-purpose *system registers*:

* The *Floating-point System ID* register (FPSID) is a read-only register whose value indicates which VFP implementation is being used. See *FPSID* on page C2-22 for details.

* The *Floating-point Status and Control* register (FPSCR) is a read/write register which provides all user-level status and control of the floating-point system. See *FPSCR* on page C2-23 for details of the FPSCR.

* The *Floating-point Exception* register (FPEXC) is a read/write register, two bits of which provide system-level status and control. The remaining bits of this register can be used to communicate exception information between the hardware and software components of the implementation, in an IMPLEMENTATION DEFINED manner. See *FPEXC* on page C2-26 for details of the FPEXC.

* Individual VFP implementations can define and use further system registers for the purpose of communicating between the hardware and software components of the implementation. All such registers are IMPLEMENTATION DEFINED. They must not be used outside the implementation itself, except as described in implementation-specific documentation.

C2.8.1   FPSID

The FPSID has the following format:

31 24	23	22 21	20	19 16	15 8	7 4	3 0
implementor	SW	format	SNG	architecture	part number	variant	revision
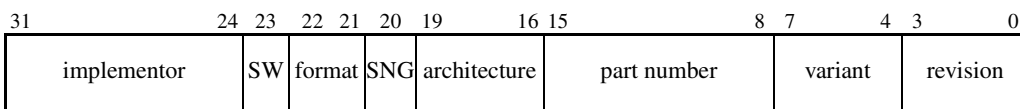
Bits[31:24]   Contain an implementor code. The following code is defined:

    0x41 = A (ARM Ltd)

All other values of the implementor code are reserved by ARM Ltd.

Bit[23]   Contains 0 if the implementation contains a hardware coprocessor, or 1 if it is a pure software implementation.

Bits[22:21]   Indicate which FSTMX/FLDMX format is used (see *Storing and reloading values of unknown precision* on page C2-17):

0b00   Indicates standard format 1.

0b01   Indicates standard format 2.

0b10   Is reserved.

0b11   Indicates a non-standard format.

Bit[20]   Contains 0 if the implementation supports both single-precision and double-precision (a D variant of the architecture), or 1 if it only supports single-precision (a non-D variant).

Bits[19:16]   Contain the architecture version number, encoded as follows:

0b0000   Indicates VFPv1.

0b0001   Indicates VFPv2.

All other values of this architecture version code are reserved by ARM Ltd.

Bits[15:8]   Contain an IMPLEMENTATION DEFINED representation of the primary part number of the VFP implementation.

Bits[7:4]   Contain an IMPLEMENTATION DEFINED variant number. This is typically used to distinguish variants of the same primary part. For example, two variants of the same VFP implementation might have hardware coprocessor interfaces designed to work with different ARM processors.

Bits[3:0]   Contain the IMPLEMENTATION DEFINED revision number of the part.

The FPSID register is read-only, and can be accessed in both privileged and unprivileged modes. Attempts to write the FPSID register are ignored.

C2.8.2 FPSCR

The FPSCR has the following format:

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
N	Z	C	V	DNM	DN	FZ	RMODE	STRIDE	DNM	LEN	IDE	DNM	IXE	UFE	OFE	DZE	IOE	IDC	DNM	IXC	UFC	OFC	DZC	IOC
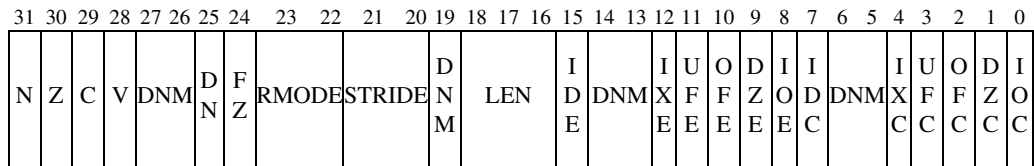
All of these bits can be read and written, and can be accessed in both privileged and unprivileged modes.

——— **Note** ———

All bits described as DNM (Do Not Modify) in the diagram are reserved for future expansion. They are initialized to zeros. Non-initialization code must use read/modify/write techniques when handling the FPSCR, to ensure that these bits are not modified. Failure to observe this rule can result in code which has unexpected side effects on future systems.

The FPSCR bits are described in the following subsections.

Condition flags

Bits[31:28] of the FPSCR contain the results of the most recent floating-point comparison:

N   Is 1 if the comparison produced a *less than* result

Z   Is 1 if the comparison produced an *equal* result

C   Is 1 if the comparison produced an *equal*, *greater than* or *unordered* result

V   Is 1 if the comparison produced an *unordered* result.

These condition flags do not directly affect conditional execution, either of ARM instructions or of VFP instructions. A comparison instruction is normally followed by an FMSTAT instruction. This transfers the FPSCR condition flags to the ARM CPSR flags, after which they can affect conditional execution.

For more details of how comparisons are performed, see *Comparison instructions* on page C3-6.

Default NaN mode control

Bit[25] is the Default NaN mode control bit. See *Default NaN mode* on page C2-14 for details.

Flush-to-zero mode control

Bit[24] of the FPSCR is the FZ bit and controls *flush-to-zero mode*. See *Flush-to-zero mode* on page C2-13 for details of this processing mode.

FZ == 0   Flush-to-zero mode is disabled and the behavior of the floating-point system is fully compliant with the IEEE 754 standard.

FZ == 1   Flush-to-zero mode is enabled.

Rounding mode control

Bits[23:22] of the FPSCR select the current rounding mode. This rounding mode is used for almost all floating-point instructions. The only floating-point instructions which do not use it are FTOSIZD, FTOSIZS, FTOUIZD and FTOUIZS, which always use RZ mode.

The rounding modes are encoded as follows:

0b00   Indicates *Round to Nearest* (RN) mode

0b01   Indicates *Round towards Plus Infinity* (RP) mode

0b10   Indicates *Round towards Minus Infinity* (RM) mode

0b11   Indicates *Round towards Zero* (RZ) mode.

See *Rounding* on page C2-9 for details of the rounding modes.

Vector length/stride control

The LEN field (bits[18:16]) of the FPSCR controls the vector length for VFP instructions that operate on short vectors, that is, how many registers are in a vector operand. Similarly, the STRIDE field (bits[21:20]) controls the vector stride, that is, how far apart the registers in a vector lie in the register bank. The allowed combinations of LEN and STRIDE are shown in Table C2-3 on page C2-25.

All other combinations of LEN and STRIDE produce UNPREDICTABLE results.

The combination LEN == 0b000, STRIDE == 0b00 is sometimes called *scalar mode*. When it is in effect, all arithmetic instructions specify simple scalar operations. Otherwise, most arithmetic instructions specify a scalar operation if their destination lies in the range S0-S7 (for single-precision) or D0-D3 (for double-precision). The full rules used to determine which operands are vectors and full details of how vector operands are specified can be found in Chapter C5 *VFP Addressing Modes* and in the individual instruction descriptions.

The rules for vector operands do not allow the same register to appear twice or more in a vector. The allowed LEN/STRIDE combinations listed in Table C2-3 on page C2-25 never cause this to happen for single-precision instructions, so single-precision scalar and vector instructions can be used with all of these LEN/STRIDE combinations.

For double-precision vector instructions, some of the allowed LEN/STRIDE combinations would cause the same register to appear twice in a vector. If a double-precision vector instruction is executed with such a LEN/STRIDE combination in effect, the instruction is UNPREDICTABLE. The last column of Table 2-2 indicates which LEN/STRIDE combinations this applies to. Double-precision scalar instructions work normally with all of the allowed LEN/STRIDE combinations.

Table C2-3 Vector length/stride combinations

LEN	STRIDE	Vector length	Vector stride	Double-precision vector instructions
0b000	0b00	1	-	All instructions are scalar
0b001	0b00	2	1	Work normally
0b001	0b11	2	2	Work normally
0b010	0b00	3	1	Work normally
0b010	0b11	3	2	UNPREDICTABLE
0b011	0b00	4	1	Work normally
0b011	0b11	4	2	UNPREDICTABLE
0b100	0b00	5	1	UNPREDICTABLE
0b101	0b00	6	1	UNPREDICTABLE
0b110	0b00	7	1	UNPREDICTABLE
0b111	0b00	8	1	UNPREDICTABLE

Input denormal flag and trap enable

Bit[15] and bit[7] of the FPSCR are the trap enable and exception bits respectively for the *Input Denormal Condition* (IDC).

The IDC flag is set whenever an operand is denormal, except when the instruction is FABS, FNEG, FCPY, FTOUI, or FTOSI. When set, the bit remains set until it is cleared by a write to the FPSCR.

When the *Input Denormal Enable* (IDE) bit is set, an instruction that causes the IDC bit to be set forces the VFP to *bounce* to the user trap handler when a trigger instruction is issued.

Exception status and control

Bits[12:8] and bits[4:0] of the FPSCR are the trap enable bits and cumulative exception bits respectively for the five types of exception. For details of what these do, see *Floating-point exceptions* on page C2-10.

Table C2-4 shows which bits are associated with each exception.

Table C2-4 Exception status and control bits

Exception type	Trap enable bit	Cumulative exception bit
Invalid Operation	IOE (bit[8])	IOC (bit[0])
Division by Zero	DZE (bit[9])	DZC (bit[1])
Overflow	OFE (bit[10])	OFC (bit[2])
Underflow	UFE (bit[11])	UFC (bit[3])
Inexact	IXE (bit[12])	IXC (bit[4])

C2.8.3   FPEXC

The FPEXC register has the following format:

```
31 30 29                                                           0
┌──┬──┬──────────────────────────────────────────────────────────┐
│EX│EN│              IMPLEMENTATION DEFINED                        │
└──┴──┴──────────────────────────────────────────────────────────┘
```
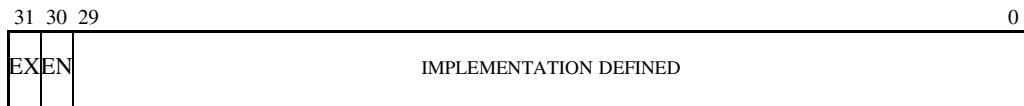
This register can only be accessed in privileged modes.

The EX bit

The EX bit (bit[31]) is a status bit which specifies how much information needs to be saved to record the state of the floating-point system. It can be read on all VFP implementations, and is mainly of interest to process swap code.

EX == 0   In this case, the only significant state in the floating-point system is the contents of the architecturally defined writable registers, that is, of the general-purpose registers, FPSCR and FPEXC. If EX == 0 when a process is swapped out, only these registers need to be saved, or reloaded when the process is swapped back in. Also, no unexpected ARM exceptions (such as an undefined instruction exception to process a pending exception in the hardware) must occur during the saving and reloading of the registers.

EX == 1   Here, there is additional IMPLEMENTATION DEFINED significant state in the floating-point system which process swap code needs to handle. This typically occurs when VFP hardware requires support code assistance to handle a potential exception, and one or more of the additional hardware system registers contains details of the potential exception. (Some

implementations describe this by saying that the hardware is in an *exceptional state*.) The actions required to swap a process out when EX == 1 and to swap such a process back in are IMPLEMENTATION DEFINED.

The behavior of the EX bit when FPEXC is written is IMPLEMENTATION DEFINED, subject to the constraint that writing a 0 to the EX bit must be a legitimate action. Otherwise, the process swap technique described above for the case EX == 0 cannot work.

The EN bit

The EN bit (bit[30]) is a global enable bit, and can be both read and written.

EN == 1   In this case, the floating-point system is enabled and operates normally.

EN == 0   Here, the floating-point system is disabled. In this state, all VFP instructions are treated as undefined instructions when executed in an unprivileged ARM processor mode, and all except the following are treated as undefined instructions when executed in a privileged ARM processor mode:

   • an FMXR to the FPEXC or FPSID register

   • an FMRX from the FPEXC or FPSID register.

─── **Note** ───

An FMXR to the FPSCR or an FMRX from the FPSCR is treated as an undefined instruction when EN == 0. If a VFP implementation contains additional system registers besides FPSID, FPSCR, and FPEXC, the behavior of FMXR instructions to them and FMRX instructions from them is IMPLEMENTATION DEFINED.

Other bits

All bits of the FPSCR other than the EX and EN bits are IMPLEMENTATION DEFINED, including whether they are readable, writable or both. They are typically used in hardware implementations for communicating exception information between the VFP hardware and its support code.

A constraint on how these bits are defined is that when the EX bit is 0, it must be possible to save and reload all significant state in the floating-point system by saving and reloading only the VFP general-purpose registers, FPSCR and FPEXC.

C2.9 Reset behavior and initialization

When a hardware VFP implementation is reset, the FPEXC EN bit is reset to 0. The behavior of all other VFP registers and of the remaining bits of FPEXC on hardware reset is IMPLEMENTATION DEFINED.

When the software component of a VFP implementation has finished initializing, the following are true:

*   The FPEXC EN bit is set to 1

*   The FPEXC EX bit is set to 0

*   All bits of the FPSCR are set to 0, with the possible exception of the condition code flags in some cases. This selects the following settings:
    —   normal IEEE 754 mode, not flush-to-zero mode
    —   the *Round to Nearest* rounding mode
    —   scalar mode (vector length 1)
    —   all exceptions are untrapped, and their cumulative status bits indicate that no exceptions of that type have been detected yet.

It is IMPLEMENTATION DEFINED whether the VFP general-purpose registers and the FPSCR condition flags are initialized, and if so, what values they are initialized to.

Chapter C3
VFP Instruction Set Overview

This chapter gives an overview of the VFP instruction set. It contains the following sections:

- *Data-processing instructions* on page C3-2
- *Load and Store instructions* on page C3-13
- *Single register transfer instructions* on page C3-17
- *Two-register transfer instructions* on page C3-21.

C3.1 Data-processing instructions

All VFP data-processing instructions are `CDP` instructions for coprocessors 10 or 11, with the following format:

31 30 29 28	27 24	23	22	21	20	19 16	15 12	11 8	7	6	5	4	3 0
cond	1 1 1 0	p	D	q	r	Fn	Fd	cp_num	N	s	M	0	Fm

p, q, r, s   These bits collectively form the instruction's primary opcode. See Table C3-1 on page C3-3 for the assignment of these opcodes. When all of p, q, r and s are 1, the instruction is a two-operand *extension instruction*, with an extension opcode specified by the Fn and N bits.

Fd and D   These bits normally specify the destination register of the instruction:

- For a single-precision instruction, Fd holds the top 4 bits of the register number and D holds the bottom bit.

- For a double-precision instruction, Fd holds the register number and D must be 0.

If D is 1 in a double-precision instruction, the instruction is UNDEFINED.

For multiply-accumulate instructions, this register is also the accumulate operand register. For comparison instructions, it is the first operand register rather than a destination register.

Fn and N   These bits normally specify the first operand register of the instruction.

- For a single-precision instruction, Fn holds the top 4 bits of the register number and N holds the bottom bit.

- For a double-precision instruction, Fn holds the register number and N must be 0.

However, if p, q, r and s are all 1, the instruction is an extension instruction, and the Fn and N fields form an extension opcode instead of specifying a register. See Table C3-2 on page C3-4 for the assignment of these extension opcodes.

If N is 1 in a double-precision non-extension instruction, the instruction is UNDEFINED.

Fm and M   These bits specify the second operand register of the instruction, or the only operand register for some extension instructions.

- For a single-precision instruction, Fm holds the top 4 bits of the register number and M holds the bottom bit.

- For a double-precision instruction, Fm holds the register number and M must be 0.

If M is 1 in a double-precision instruction, the instruction is UNDEFINED.

cp_num   If cp_num is 0b1010 (coprocessor number 10), the instruction is a single-precision instruction. If cp_num is 0b1011 (coprocessor number 11), the instruction is a double-precision instruction.

For the instructions that convert between single-precision and double-precision (`FCVTDS` and `FCVTSD`), cp_num matches the source precision.

Table C3-1 and Table C3-2 on page C3-4 show the assignment of VFP data-processing opcodes. In these tables, Fd is used to mean a destination register of the appropriate precision, that is, Sd for single-precision instructions and Dd for double-precision instructions. Fn and Fm are used similarly.

Table C3-1 VFP data-processing primary opcodes

p	q	r	s	Instruction name cp_num=10	Instruction name cp_num=11	Instruction functionality
0	0	0	0	FMACS	FMACD	Fd = Fd + (Fn * Fm)
0	0	0	1	FNMACS	FNMACD	Fd = Fd - (Fn * Fm)
0	0	1	0	FMSCS	FMSCD	Fd = -Fd + (Fn * Fm)
0	0	1	1	FNMSCS	FNMSCD	Fd = -Fd - (Fn * Fm)
0	1	0	0	FMULS	FMULD	Fd = Fn * Fm
0	1	0	1	FNMULS	FNMULD	Fd = -(Fn * Fm)
0	1	1	0	FADDS	FADDD	Fd = Fn + Fm
0	1	1	1	FSUBS	FSUBD	Fd = Fn - Fm
1	0	0	0	FDIVS	FDIVD	Fd = Fn / Fm
1	0	0	1	-	-	UNDEFINED
1	0	1	0	-	-	UNDEFINED
1	0	1	1	-	-	UNDEFINED
1	1	0	0	-	-	UNDEFINED
1	1	0	1	-	-	UNDEFINED
1	1	1	0	-	-	UNDEFINED
1	1	1	1	See Table C3-2 on page C3-4	See Table C3-2 on page C3-4	Extension instructions

Table C3-2 VFP data-processing extension opcodes

Extension opcode		Instruction name		
Fn	**N**	**cp_num=10**	**cp_num=11**	**Instruction functionality**
0000	0	FCPYS	FCPYD	Fd = Fm
0000	1	FABSS	FABSD	Fd = abs(Fm)
0001	0	FNEGS	FNEGD	Fd = -Fm
0001	1	FSQRTS	FSQRTD	Fd = sqrt(Fm)
001x	x	-	-	UNDEFINED
0100	0	FCMPS	FCMPD	Compare Fd with Fm, no exceptions on quiet NaNs
0100	1	FCMPES	FCMPED	Compare Fd with Fm, with exceptions on quiet NaNs
0101	0	FCMPZS	FCMPZD	Compare Fd with 0, no exceptions on quiet NaNs
0101	1	FCMPEZS	FCMPEZD	Compare Fd with 0, with exceptions on quiet NaNs
0110	x	-	-	UNDEFINED
0111	0	-	-	UNDEFINED
0111	1	FCVTDS	FCVTSD	Single ↔ double-precision conversions
1000	0	FUITOS	FUITOD	Unsigned integer → floating-point conversions
1000	1	FSITOS	FSITOD	Signed integer → floating-point conversions
1001	x	-	-	UNDEFINED
101x	x	-	-	UNDEFINED
1100	0	FTOUIS	FTOUID	Floating-point → unsigned integer conversions
1100	1	FTOUIZS	FTOUIZD	Floating-point → unsigned integer conversions, RZ mode
1101	0	FTOSIS	FTOSID	Floating-point → signed integer conversions
1101	1	FTOSIZS	FTOSIZD	Floating-point → signed integer conversions, RZ mode
111x	x	-	-	UNDEFINED

C3.1.1 Basic arithmetic instructions and square root

The `FADDS`, `FSUBS`, `FMULS`, `FDIVS`, and `FSQRTS` instructions provide the four basic arithmetic operations and square root on single-precision values. Similarly, the `FADDD`, `FSUBD`, `FMULD`, `FDIVD`, and `FSQRTD` instructions supply these operations on double-precision values. In addition, the `FNMULS` and `FNMULD` instructions supply negated multiplications in single and double-precision respectively. Their results are precisely equivalent to those of performing an `FMULS` or `FMULD` instruction followed by an `FNEGS` or `FNEGD` instruction (which inverts the sign of the result).

All of these instructions can be made to operate on short vectors by setting the FPSCR LEN and STRIDE fields appropriately (see Chapter C5 *VFP Addressing Modes* for details).

The operations performed by all these instructions are always treated as floating-point operations, both for NaN handling and flush-to-zero mode. In particular, signaling NaN operands cause Invalid Operand exceptions, and in flush-to-zero mode, denormalized operands are treated as +0 and sufficiently small results are forced to +0.

C3.1.2 Multiply-accumulate instructions

`FMACS`, `FMACD`, `FNMACS`, `FNMACD`, `FMSCS`, `FMSCD`, `FNMSCS`, and `FNMSCD` are *multiply-accumulate* instructions. They multiply their two main operands, possibly invert the sign bit of the product, add or subtract the value in the destination register and write the result back to the destination register. They are in all respects equivalent to the following sequences of basic arithmetic and negation instructions:

```
FMACS   Sd,Sn,Sm:    FMULS    St,Sn,Sm
                     FADDS    Sd,St,Sd

FMACD   Dd,Dn,Dm:    FMULD    Dt,Dn,Dm
                     FADDD    Dd,Dt,Dd

FNMACS  Sd,Sn,Sm:    FMULS    St,Sn,Sm
                     FNEGS    St,St
                     FADDS    Sd,St,Sd

FNMACD  Dd,Dn,Dm:    FMULD    Dt,Dn,Dm
                     FNEGD    Dt,Dt
                     FADDD    Dd,Dt,Dd

FMSCS   Sd,Sn,Sm:    FMULS    St,Sn,Sm
                     FSUBS    Sd,St,Sd

FMSCD   Dd,Dn,Dm:    FMULD    Dt,Dn,Dm
                     FSUBD    Dd,Dt,Dd

FNMSCS  Sd,Sn,Sm:    FMULS    St,Sn,Sm
                     FNEGS    St,St
                     FSUBS    Sd,St,Sd

FNMSCD  Dd,Dn,Dm:    FMULD    Dt,Dn,Dm
                     FNEGD    St,St
                     FSUBD    Dd,Dt,Dd
```

where St or Dt describes a notional register used to hold intermediate results, treated as being a scalar if Sd or Dd is a scalar and a vector if Sd or Dd is a vector.

———— **Note** ————

This implies that each multiply-accumulate operation involves two roundings:

- one on the multiplication result
- one on the result of the final addition or subtraction.

Both of these roundings are performed fully and as defined by the IEEE 754 standard. In particular, these instructions do not specify *fused multiply-accumulates* as used in a number of other architectures.

All of these instructions can be made to operate on short vectors by setting the FPSCR LEN and STRIDE fields appropriately (see Chapter C5 *VFP Addressing Modes* for details). The operations performed by all these instructions are always treated as floating-point operations, both for NaN handling and flush-to-zero mode. In particular, signaling NaN operands cause Invalid Operand exceptions, and in flush-to-zero mode, denormalized operands are treated as +0 and sufficiently small results are forced to +0.

C3.1.3   Comparison instructions

The FCMPS, FCMPD, FCMPES, and FCMPED instructions perform comparisons between two register values. The FCMPZS, FCMPZD, FCMPEZS, and FCMPEZD instructions perform comparisons between a register value and the constant +0.

The IEEE 754 standard specifies that precisely one of four relationships holds between any two values being compared. These are as follows:

- Two values are considered equal if any of the following conditions holds:
  — They are both numeric and have the same numerical value. This usually means that they have precisely the same representation, but also includes the case that one is +0 and the other is -0.
  — They are both +∞ (plus infinity).
  — They are both −∞ (minus infinity).

- The first value is considered less than the second value if any of the following conditions holds:
  — They are both numeric and the numeric value of the first is less than that of the second.
  — The first is −∞ (minus infinity) and the second is numeric.
  — The first is numeric and the second is +∞ (plus infinity).
  — The first is −∞ (minus infinity) and the second is +∞ (plus infinity).

- The first value is considered greater than the second value if any of the following conditions holds:
  — They are both numeric and the numeric value of the first is greater than that of the second.
  — The first is +∞ (plus infinity) and the second is numeric.
  — The first is numeric and the second is −∞ (minus infinity).
  — The first is +∞ (plus infinity) and the second is −∞ (minus infinity).

- Two values are *unordered* if either or both of them are NaNs.

——— **Note** ———

If both values are the same NaN, the comparison result is *unordered*, not *equal*. If an exact bit-by-bit comparison is wanted, the ARM comparison instructions must be used rather than VFP comparison instructions, both for this reason and because +0 and -0 compare as equal.

For all the comparison instructions, the result of the comparison is placed in the FPSCR flags, as shown in Table C3-3:

Table C3-3 VFP comparison flag values

Comparison result	N	Z	C	V
Equal	0	1	1	0
Less than	1	0	0	0
Greater than	0	0	1	0
Unordered	0	0	1	1

These FPSCR flag values need to be copied to the ARM CPSR flags before ARM conditional execution can be based on them. For this purpose, a special form of the FMRX instruction (called FMSTAT) is used. This is described in *System register transfer instructions* on page C3-20.

When the result of the comparison is *unordered*, it is possible that the comparison can also generate an Invalid Operation exception because of the NaN operand(s). These instructions supply two distinct forms of Invalid Operation exception generation:

- The FCMPS, FCMPD, FCMPZS, and FCMPZD instructions have the normal behavior of generating an Invalid Operation exception when either or both of their operands are signaling NaNs. If neither operand is a signaling NaN, but one or both are quiet NaNs, they generate an *unordered* result without an accompanying Invalid Operation exception.

- The FCMPES, FCMPED, FCMPEZS, and FCMPEZD instructions generate an Invalid Operation exception when either or both of their operands are NaNs, regardless of whether they are signaling or quiet NaNs. It is not possible to get an *unordered* result from these instructions without an accompanying Invalid Operation exception.

The VFP comparison instructions always treat their operands as scalars, regardless of the settings of the FPSCR LEN and STRIDE fields.

The operations performed by all these instructions are always treated as floating-point operations, both for NaN handling and flush-to-zero mode. In particular, signaling NaN operands cause Invalid Operand exceptions, and in flush-to-zero mode, denormalized operands are treated as +0.

Testing the IEEE 754 predicates

The IEEE 754 standard specifies two ways in which a floating-point comparison can deliver its results:

*   As a *condition code* result, identifying one of the four relations:

    —   equal

    —   less than

    —   greater than

    —   unordered.

*   As a true-or-false result to one of twenty-six *predicates*, each of which specifies a particular test on the values. Six of these are the standard ==, !=, <, <=, > and >= comparisons, used in common languages like C, C++ and related languages.

The VFP architecture uses the first approach. However, its condition code results have been carefully chosen to allow ARM conditional execution to test as many of the predicates as possible after a sequence of a VFP comparison instruction and an FMSTAT instruction. This includes all six of the commonly-used predicates.

Table C3-4 shows how each predicate must be tested to get the correct results according to the IEEE 754 standard:

Table C3-4 VFP predicate testing

Common language condition	IEEE predicate	Instruction type	ARM condition
==	=	FCMP	EQ
!=	?<>	FCMP	NE
>	>	FCMPE	GT
>=	>=	FCMPE	GE
<	<	FCMPE	MI or CC
<=	<=	FCMPE	LS
	?	FCMP	VS
	<>	FCMPE	Two conditions
	<=>	FCMPE	VC
	?>	FCMP	HI
	?>=	FCMP	PL or CS
	?<	FCMP	LT

Table C3-4 VFP predicate testing (continued)

Common language condition	IEEE predicate	Instruction type	ARM condition
	?<=	FCMP	LE
	?=	FCMP	Two conditions
	NOT(>)	FCMPE	LE
	NOT(>=)	FCMPE	LT
	NOT(<)	FCMPE	PL or CS
	NOT(<=)	FCMPE	HI
	NOT(?)	FCMP	VC
	NOT(<>)	FCMPE	Two conditions
	NOT(<=>)	FCMPE	VS
	NOT(?>)	FCMP	LS
	NOT(?>=)	FCMP	MI or CC
	NOT(?<)	FCMP	GE
	NOT(?<=)	FCMP	GT
	NOT(?=)	FCMP	Two conditions

In each case, the two main choices to be made are:

- Whether to use an FCMP-type instruction (that is, the appropriate one of FCMPS, FCMPD, FCMPZS or FCMPZD) or an FCMPE-type instruction (the appropriate one of FCMPES, FCMPED, FCMPEZS or FCMPEZD). This choice causes the predicate to have the correct behavior with regard to Invalid Operation exceptions.

- Which ARM condition is to be used. This is not always obvious. For example, a standard < comparison on floating-point numbers must use the ARM MI or LO/CC condition, not LT, despite the fact that floating-point comparisons are always signed.

  If this column contains *two conditions*, no single ARM condition can be used to test the predicate. Each of these predicates can be tested using a suitable combination of two ARM conditions, in several different ways. For example, the <> predicate can be tested by checking that NE and VC are both true, or that either of GT and MI is true.

C3.1.4   Conversion instructions

All of the VFP conversion instructions always treat their operands as scalars, regardless of the settings of the FPSCR LEN and STRIDE fields.

Conversions between single and double-precision

The FCVTDS and FCVTSD instructions perform conversions between single-precision and double-precision values. FCVTDS converts single-precision to double-precision and is a coprocessor 10 instruction, while FCVTSD converts double-precision to single-precision and is a coprocessor 11 instruction.

The FCVTDS and FCVTSD conversions are always treated as floating-point operations, both for NaN handling and flush-to-zero mode. In particular, signaling NaN operands cause Invalid Operand exceptions, and in flush-to-zero mode, denormalized operands are treated as +0.

The only exception possible for FCVTDS is an Invalid Operation exception caused by a signaling NaN operand, as single-precision numbers can always be represented exactly in double-precision. FCVTSD can additionally generate Overflow, Underflow and/or Inexact exceptions.

Conversions from floating-point to integers

The FTOSIS and FTOSID instructions convert floating-point values to signed integers, and the FTOUIS and FTOUID instructions convert floating-point values to unsigned integers, using the rounding mode specified by the FPSCR.

Variants of these instructions called FTOSIZS, FTOSIZD, FTOUIZS, and FTOUIZD perform similar conversions, but using Round towards Zero mode. These are useful because C and related languages specify that floating-point $\rightarrow$ integer conversions use this mode, whereas almost all other operations normally use Round to Nearest mode. Using these instructions avoids the need to change the FPSCR rounding mode every time a floating-point $\rightarrow$ integer conversion is wanted.

All of the floating-point $\rightarrow$ integer conversion instructions place their integer result in a single-precision register. This result can then be used in any of the following ways:

- store it to memory using FSTS or FSTMS
- transfer it to an ARM register using FMRS
- convert it to a floating-point number using any of FSITOS, FSITOD, FUITOS or FUITOD.

The operations performed by all these instructions are always treated as floating-point operations, both for NaN handling and flush-to-zero mode. In particular, signaling NaN operands cause Invalid Operand exceptions, and in flush-to-zero mode, denormalized operands are treated as +0.

Most exceptional conditions that can occur during these instructions are signaled as Invalid Operation exceptions. These cannot produce the normal quiet NaN value as their result, as the destination is an integer. Instead, the following list of values that generate Invalid Operation exceptions also specifies the integer default result in each case:

- If the operand is numeric, but converting it to an integer using the appropriate rounding mode would produce an integer that is greater than the maximum possible destination integer, the default result is the maximum possible destination integer.

- If the operand is numeric, but converting it to an integer using the appropriate rounding mode would produce an integer that is less than the minimum possible destination integer, the default result is the minimum possible destination integer.

- If the operand is +∞ (plus infinity), the default result is the maximum possible destination integer.

- If the operand is −∞ (minus infinity), the default result is the minimum possible destination integer.

- If the operand is a NaN (either signaling or quiet), the default result is 0.

Apart from these Invalid Operation exceptions, the only exceptions that can be produced by the floating-point → integer conversions are Inexact exceptions.

Conversions from integers to floating-point

The `FSITOS` and `FSITOD` instructions convert signed integers to floating-point values, and the `FUITOS` and `FUITOD` instructions convert unsigned integers to floating-point values. All of them take their integer operand from a single-precision register. This operand can have been placed in the register earlier in any of the following ways:

- loading it from memory using `FLDS` or `FLDMS`

- transferring it from an ARM register using `FMSR`

- converting a floating-point number to an integer using any of `FTOSIS`, `FTOSID`, `FTOSIZS`, `FTOSIZD`, `FTOUIS`, `FTOUID`, `FTOUIZS`, or `FTOUIZD`.

When an integer 0 is converted to floating-point, the result is +0. For the `FSITOS` and `FUITOS` instructions, some integer operands that exceed 2^{24} in magnitude cannot be converted exactly. Conversions of these operands are rounded according to the rounding mode specified in the FPSCR, with an Inexact exception being generated. Otherwise, no exceptions are possible with the integer → floating-point conversions.

C3.1.5   Copy, negation and absolute value instructions

The `FCPYS` and `FCPYD` instructions perform an exact copy of a floating-point value from one register to another.

The `FNEGS` and `FNEGD` instructions do the same as `FCPYS` and `FCPYD`, except that they invert the sign bit during the copy. This negates numerical values and infinities, in the way described in the Appendix to the IEEE 754 standard.

The `FABSS` and `FABSD` instructions do the same as `FCPYS` and `FCPYD`, except that they change the sign bit to 0 during the copy. This takes the absolute value of numerical values and infinities, in the way described in the Appendix to the IEEE 754 standard.

All of these instructions can be made to operate on short vectors by setting the FPSCR LEN and STRIDE fields appropriately (see Chapter C5 *VFP Addressing Modes*).

The IEEE 754 standard and its Appendix allow all these operations to be treated as non floating-point operations with regard to NaN handling. The VFP architecture requires this to be done. In particular, this implies the following:

- The VFP architecture requires these instructions *not* to generate Invalid Operation when their operands are signaling NaNs.

- The results of these instructions are generated by copying their operands (with appropriate sign bit adjustments), even when their operands are NaNs. This overrides the normal rules for generating the results of instructions with one or more NaN operands (described in *NaNs* on page C2-5).

In addition, the VFP architecture requires these instructions to be treated as non floating-point operations with regard to flush-to-zero mode. In flush-to-zero mode, they copy denormalized operands in the same way as they do in normal mode, and do not treat the operands as +0.

——— **Note** ———

Calculating the value of -x using FNEGS or FNEGD does not produce exactly the same results as calculating either (+0 - x) or (-0 - x) using FSUBS or FSUBD. The differences are:

- FSUBS or FSUBD produces an Invalid Operation exception if x is a signaling NaN, whereas FNEGS or FNEGD produces x with its sign bit inverted, without an exception.

- FSUBS or FSUBD produces an exact copy of x if x is a quiet NaN, whereas FNEGS or FNEGD produces x with its sign bit inverted.

- FNEGS or FNEGD applied to a zero always produces an oppositely signed zero. Calculating the value of (+0 - x) using FSUBS or FSUBD does this in RM rounding mode, but always produces +0 in RN, RP or RZ rounding mode. Calculating (-0 - x) always produces -0 in RM rounding mode, and produces an oppositely signed zero in RN, RP or RZ rounding mode.

- In flush-to-zero mode, the calculation using FSUBS or FSUBD treats denormalized operands as +0, and therefore produce a zero result if x is denormalized. FNEGS or FNEGD ignore flush-to-zero mode and produce a result of x with its sign bit inverted.

———

C3.2 Load and Store instructions

All VFP Load and Store instructions are `LDC` and `STC` instructions respectively for coprocessors 10 and 11, with the following format:

31 30 29 28	27 25	24	23	22	21	20	19 16	15 12	11 8	7 6 5 4 3 0
cond	1 1 0	P	U	D	W	L	Rn	Fd	cp_num	offset

P, U, W These bits specify an addressing mode for the `LDC` or `STC` instruction, as described in *ARM Addressing Mode 5 - Load and Store Coprocessor* on page A5-60. In addition, a VFP implementation uses them to determine which load/store operation is required, as shown in Table C3-5 on page C3-14.

Fd and D These bits specify the destination floating-point register of a load instruction, or the source floating-point register of a store instruction.

- For a single-precision instruction, Fd holds the top 4 bits of the register number and D holds the bottom bit.

- For a double-precision instruction, Fd holds the register number and D must be 0.

If D is 1 in a double-precision instruction, the instruction is UNDEFINED.

For Load Multiple and Store Multiple instructions, the register specified by these fields is the lowest-numbered register to be transferred. Subsequent registers are transferred in order of register number, up to the number of registers determined by the offset field. If this would result in a register after S31 or D15 being transferred, the results are UNPREDICTABLE.

L bit This bit determines whether the instruction is a load (L == 1) or a store (L == 0).

Rn This specifies the ARM register used as the base register for the address calculation, as described in *ARM Addressing Mode 5 - Load and Store Coprocessor* on page A5-60.

cp_num If cp_num is 0b1010 (coprocessor number 10), the instruction is a single-precision instruction. If cp_num is 0b1011 (coprocessor number 11), the instruction is either a double-precision instruction or one of the instructions used to handle values of unknown precision (see *Storing and reloading values of unknown precision* on page C2-17).

offset These bits specify the word offset which is applied to the base register value to obtain the starting memory address for the transfer, as described in *ARM Addressing Mode 5 - Load and Store Coprocessor* on page A5-60.

The least significant bit of this offset also helps to determine which load/store operation is required, as shown in Table C3-5 on page C3-14. In addition, for Load Multiple and Store Multiple instructions, the offset determines how many registers are to be transferred.

Table C3-5 on page C3-14 shows how the name and other details of the instruction are determined from the P, U, W, and L bits and the cp_num and offset fields:

Table C3-5 VFP load and store instructions

PUW	cp num	offset [0]	Instruction L==0	Instruction L==1	Addr mode	Registers transferred
0 0 x	x	x	UNDEFINED	-	-	-
0 1 0	0b1010	x	FSTMS	FLDMS	Unindexed	(offset) single-precision registers
0 1 0	0b1011	0	FSTMD	FLDMD	Unindexed	(offset)/2 double-precision registers
0 1 0	0b1011	1	FSTMX	FLDMX	Unindexed	(offset-1)/2 double-precision registers
0 1 1	0b1010	x	FSTMS	FLDMS	Increment	(offset) single-precision registers
0 1 1	0b1011	0	FSTMD	FLDMD	Increment	(offset)/2 double-precision registers
0 1 1	0b1011	1	FSTMX	FLDMX	Increment	(offset-1)/2 double-precision registers
1 0 0	0b1010	x	FSTS	FLDS	Negative offset	One single-precision register
1 0 0	0b1011	x	FSTD	FLDD	Negative offset	One double-precision register
1 0 1	0b1010	x	FSTMS	FLDMS	Decrement	(offset) single-precision registers
1 0 1	0b1011	0	FSTMD	FLDMD	Decrement	(offset)/2 double-precision registers
1 0 1	0b1011	1	FSTMX	FLDMX	Decrement	(offset-1)/2 double-precision registers
1 1 0	0b1010	x	FSTS	FLDS	Positive offset	One single-precision register
1 1 0	0b1011	x	FSTD	FLDD	Positive offset	One double-precision register
1 1 1	x	x	UNDEFINED	-	-	-

All load instructions perform a copy of the loaded value(s) from memory, and all store instructions perform a copy of the stored value(s) to memory. No exceptions are ever raised and the value(s) transferred are not changed, except possibly for a reversible conversion to the internal register format of an implementation. The copy is treated as a non floating-point operation for the purposes of NaN handling and FTZ mode. In particular, the VFP architecture requires:

- a load or store of a signaling NaN not to raise an Invalid Operation exception, nor to change the signaling NaN into a quiet NaN

- a load or store of a denormalized number in flush-to-zero mode not to change it into +0.

C3.2.1   Load/store one value

The FLDS and FSTS instructions allow single-precision values and 32-bit integers to be loaded and stored, and the FLDD and FSTD instructions allow double-precision values to be loaded and stored. Each of these instructions transfers just one register of the type concerned.

Of the addressing modes described in *ARM Addressing Mode 5 - Load and Store Coprocessor* on page A5-60, only the Immediate offset mode (see *Load and Store Coprocessor - Immediate offset* on page A5-62) is allowed for these instructions. This addressing mode allows the address to be specified by the base register value Rn, plus or minus an immediate offset which lies in the range 0 to 1020 and is a multiple of 4. No base register write-back is available.

C3.2.2   Load/store multiple values

The FLDMS and FSTMS instructions allow multiple single-precision values and/or integers to be loaded and stored, and the FLDMD and FSTMD instructions allow multiple double-precision values to be loaded and stored.

Each of these instructions transfers a number of registers determined by the offset field of the instruction. The offset field is equal to the total number of words transferred for all of these instructions, that is, it is the number of registers for FLDMS and FSTMS, and twice the number of registers for FLDMD and FSTMD.

In addition, the FSTMX instruction can be used to store double-precision registers when it is not known whether they contain single-precision or double-precision values, in a format that allows a matching FLDMX instruction to reload them correctly (see *Storing and reloading values of unknown precision* on page C2-17). In these instructions, the offset field is twice the number of double-precision registers to be transferred, plus one. This is the maximum number of words these instructions can transfer. Some implementations transfer one fewer word than this maximum, leaving a memory word unused.

The FSTMX and FLDMX instructions are encoded as coprocessor 11 instructions, like FSTMD and FLDMD. They are distinguished from the latter by the fact that the offset field is odd in FSTMX and FLDMX instructions, and even in FSTMD and FLDMD instructions.

The FSTMX and FLDMX instructions are the only coprocessor 11 instructions which are present in single-precision-only variants (non-D variants) of the VFP architecture. To aid software portability, it is recommended that programs written for such variants must use them in the same situations as a program written for a D variant would, even though the registers are known to hold single-precision values in non-D variants. The main situations affected are when storing and reloading callee-save registers, and in process swap code.

Three addressing modes are available for these instructions:

*   Unindexed mode is the same as the LDC/STC Unindexed addressing mode (see *Load and Store Coprocessor - Unindexed* on page A5-68). The base register Rn determines the starting address for the transfer and is left unchanged.

    The offset field determines the number of registers to transfer, but does not affect the address calculations.

- Increment mode is the same as the `LDC`/`STC` Immediate post-indexed addressing mode with a positive offset (see *Load and Store Coprocessor - Immediate post-indexed* on page A5-66). The base register Rn determines the starting address for the transfer. The offset field determines the number of registers to transfer, and is also multiplied by 4, added to the value of Rn and written back to Rn.

  After the transfer, Rn therefore points to the memory word immediately after the last word to be transferred (or the last word that *could* have been transferred in the case of `FSTMX` and `FLDMX`). This means that it is suitable for pushing values on to an Empty Ascending stack or for popping them from a Full Descending stack.

- Decrement mode is the same as the `LDC`/`STC` Immediate pre-indexed addressing mode with a negative offset (see *Load and Store Coprocessor - Immediate pre-indexed* on page A5-64). The offset is multiplied by 4 and added to the value of the base register Rn to determine the starting address for the transfer, and this starting address is written back to Rn. The offset field also determines the number of registers to transfer.

  Before the transfer, Rn therefore points to the memory word immediately after the last word to be transferred (or the last word that *could* have been transferred in the case of `FSTMX` and `FLDMX`). This means that it is suitable for pushing values on to a Full Descending stack or for popping them from an Empty Ascending stack.

—— **Note** ——

There are no short vector forms of the load and store instructions as such, but the `FLDMS`, `FLDMD`, `FSTMS` and `FSTMD` instructions can be used to load and store many of the possible short vectors. However, note that short vectors wrap around within banks as described in Chapter C5 *VFP Addressing Modes*, while the load multiple and store multiple instructions simply advance linearly through S0-S31 or D0-D15. If a short vector that wraps around is to be loaded or stored, two or more instructions are needed.

C3.3 Single register transfer instructions

All VFP single-register transfer instructions are MCR and MRC instructions for coprocessors 10 and 11, with the following format:

31 30 29 28	27 24	23 21	20	19 16	15 12	11 8	7	6 5 4	3 0
cond	1 1 1 0	opcode	L	Fn	Rd	cp_num	N	0 0 1	SBZ

opcode   This determines which register transfer operation is required, as shown in Table C3-6 on page C3-18.

L bit   This bit determines the direction of the transfer:

L == 0   From an ARM register to a VFP register. (An MCR instruction.)

L == 1   From a VFP register to an ARM register. (An MRC instruction.)

Fn and N bit   These bits specify the VFP register involved in the transfer:

- For a single-precision register, Fn holds the top 4 bits of the register number, and N holds the bottom bit.

- For a double-precision register, Fn holds the register number, and N must be 0.

- For a system register, Fn and N specify the register as shown in Table C3-7 on page C3-18.

If N is 1 in an instruction that transfers a double-precision register, the instruction is UNDEFINED.

Rd   This specifies the ARM register involved in the transfer. If Rd is R15, the behavior is as specified for the generic ARM instruction:

- For an MCR instruction (L == 0), the instruction is UNPREDICTABLE.

- For an MRC instruction (L == 1), the top 4 bits of the value transferred are placed in the ARM condition code flags, and the remaining 28 bits are discarded. The FMSTAT instruction uses this behavior to transfer comparison results to the ARM.

cp_num   If cp_num is 0b1010 (coprocessor number 10), the instruction is a single-precision or system register transfer.

If cp_num is 0b1011 (coprocessor number 11), the instruction is a double-precision register transfer.

Table C3-6 shows the assignment of register transfer opcodes and other details of the instructions:

Table C3-6 VFP single register transfer instructions

opcode	cp_num	L	Instruction name	Instruction functionality
000	0b1010	0	FMSR	Sn = Rd
000	0b1010	1	FMRS	Rd = Sn
000	0b1011	0	FMDLR	Dn[31:0] = Rd
000	0b1011	1	FMRDL	Rd = Dn[31:0]
001	0b1010	x	-	UNDEFINED
001	0b1011	0	FMDHR	Dn[63:32] = Rd
001	0b1011	1	FMRDH	Rd = Dn[63:32]
01x	0b101x	x	-	UNDEFINED
10x	0b101x	x	-	UNDEFINED
110	0b101x	x	-	UNDEFINED
111	0b1010	0	FMXR	SystemReg(Fn,N) = Rd
111	0b1010	1	FMRX	Rd = SystemReg(Fn,N)
111	0b1011	x	-	UNDEFINED

Table C3-7 shows how system registers are encoded in FMXR and FMRX instructions:

Table C3-7 VFP system register encodings

Fn	N	System register
0b0000	0	FPSID
0b0001	0	FPSCR
0b1000	0	FPEXC

Encodings that are not shown in this table are:

- Reserved for future expansion if the top bit of Fn is 0. FMXR and FMRX instructions using these encodings are UNDEFINED.

- Reserved for additional IMPLEMENTATION DEFINED system registers if the top bit of Fn is 1. FMXR and FMRX instructions using these encodings are IMPLEMENTATION DEFINED.

ARM DDI 0100F-02

C3.3.1 General-purpose single register transfer instructions

The FMRS instruction allows a single-precision value or a 32-bit integer in a single-precision register to be transferred to an ARM register, and the FMSR instruction allows a similar transfer from an ARM register to a single-precision register.

The FMRDH and FMRDL instructions allow a double-precision value in a double-precision register to be transferred to a pair of ARM registers. The FMRDH instruction transfers the most significant word of the double-precision value, which contains the sign, exponent and 20 most significant fraction bits. The FMRDL instruction transfers the least significant word, which contains the remaining fraction bits.

Similarly, the FMDHR and FMDLR instructions allow a double-precision value in a pair of ARM registers to be transferred to a double-precision register. FMDHR transfers the most significant word and FMDLR transfers the least significant word.

——— **Note** ———

The FMDHR and FMDLR instructions must be used in pairs, writing to the same double-precision register. These need not be executed consecutively, but while one of a pair has been executed and the other has not, the only valid uses of the destination double-precision register are:

*   as the destination register of the second instruction of the pair

*   storing it with FSTMX and reloading it with FLDMX, and using it for other purposes between the store and the reload.

All of these instructions always treat their floating-point operand as a scalar, regardless of the settings of the FPSCR LEN and STRIDE fields.

The register transfer performed is always a simple copy. No exceptions are ever raised and the value transferred is not changed, except possibly for a reversible conversion to or from the internal register format of an implementation.

The copy is treated as a non floating-point operation for the purposes of NaN handling and flush-to-zero mode. In particular, the VFP architecture requires:

*   a load or store of a signaling NaN not to raise an Invalid Operation exception, nor to change the signaling NaN into a quiet NaN

*   a load or store of a denormalized number in flush-to-zero mode not to change it into +0.

C3.3.2   System register transfer instructions

The FMRX instruction transfers a system register value to an ARM register, and the FMXR instruction transfers an ARM register value to a system register. Their exact effects depend on the definition of the system register concerned. For more details, see *System registers* on page C2-21 for the architecturally defined system registers, or implementation documentation for IMPLEMENTATION DEFINED system registers.

These instructions are *serializing* instructions. When one of them is executed, the register transfer is delayed until:

- all floating-point operations in progress have determined whether they are going to generate an exception

- any trapped exception handling or other software processing of floating-point operations in progress has completed

- all effects of floating-point operations in progress on system register contents (such as setting cumulative exception flags for untrapped exceptions) have occurred

- all floating-point operations in progress are no longer affected by changes to system register contents (for example, by rounding mode or flush-to-zero mode changes).

ARM DDI 0100F-02

C3.4 Two-register transfer instructions

All VFP two-register transfer instructions are MCRR and MRRC instructions for coprocessors 10 and 11, with the following format:

31 30 29 28	27 24	23 21	20	19 16	15 12	11 8	7	6	5	4	3 0
cond	1 1 0 0	0 1 0	L	Rn	Rd	cp_num	0	R	M	1	Fm

L bit   This bit determines the direction of the transfer:

  L == 0   From two ARM registers to a VFP register. (An MCRR instruction.)

  L == 1   From a VFP register to an ARM register. (An MRRC instruction.)

Fm and M bit

  These bits specify the VFP register, or register pair, involved in the transfer:

  • For a pair of single-precision registers, Fm holds the top four bits of the register number, and M holds the bottom bit.

  • For a double-precision register, Fm holds the register number, and M must be 0.

  If M is 1 in an instruction that transfers a double-precision register, the instruction is UNDEFINED.

Rn   Specifies the ARM register for the upper half of a double-precision register, or for the Fm single-precision VFP register.

  If Rn is R15, the behavior is UNPREDICTABLE.

Rd   Specifies the ARM register for the lower half of a double-precision register, or for the (Fm+1) single-precision VFP register.

  If Rd is R15, the behavior is UNPREDICTABLE.

cp_num   If cp_num is 0b1010 (coprocessor number 10), the instruction is two single-precision register transfers.

  If cp_num is 0b1011 (coprocessor number 11), the instruction is a double-precision register transfer.

Table C3-8 shows details of the instructions:

Table C3-8 VFP two register transfer instructions

cp_num	L	Instruction name	Instruction functionality
0b1010	0	FMSRR	Fm = Rn, (Fm+1) = Rd
0b1010	1	FMRRS	Rn = Fm, Rd = (Fm+1)
0b1011	0	FMDRR	Fm[31:0] = Rd, Fm[63:32] = Rn
0b1011	1	FMRRD	Rd = Fm[31:0], Rn = Fm[63:32]

C3.4.1 Two-register transfer instructions

The FMRRS instruction allows two single-precision values, or 32-bit integers, in single-precision registers to be transferred to two ARM registers. The FMSRR instruction allows a similar transfer from two ARM registers to two single-precision registers.

The FMRRD instruction allows a double-precision value in a double-precision register to be transferred to a pair of ARM registers.

Similarly, the FMDRR instruction allows a double-precision value in a pair of ARM registers to be transferred to a double-precision register.

All of these instructions always treat their floating-point operand as a scalar, regardless of the settings of the FPSCR LEN and STRIDE fields.

The register transfer performed is always a simple copy. No exceptions are ever raised and the value transferred is not changed, except possibly for a reversible conversion to or from the internal register format of an implementation.

The copy is treated as a non floating-point operation for the purposes of NaN handling and flush-to-zero mode. In particular, the VFP architecture requires:

- a load or store of a signaling NaN not to raise an Invalid Operation exception, nor to change the signaling NaN into a quiet NaN

- a load or store of a denormalized number in FTZ mode not to change it into +0.

Chapter C4
VFP Instructions

This chapter describes the syntax and usage of each VFP instruction, in the section:

- *Alphabetical list of VFP instructions* on page C4-2.

C4.1 Alphabetical list of VFP instructions

Each VFP instruction is described in detail on the following pages.

C4.1.1 FABSD

31	28	27	26	25	24	23	22	21	20	19	18	17	16	15	12	11	10	9	8	7	6	5	4	3	0
cond		1	1	1	0	1	0	1	1	0	0	0	0	Dd		1	0	1	1	1	1	0	0	Dm	

The FABSD (Floating-point Absolute Value, Double-precision) instruction writes the absolute value of a double-precision register to another double-precision register. It can also perform a vector version of this operation.

Syntax

```
FABSD{<cond>}  <Dd>, <Dm>
```

where:

<cond>   Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-7. If <cond> is omitted, the AL (always) condition is used.

<Dd>   Specifies the destination register.

<Dm>   Specifies the source register.

Architecture version

D variants only

Exceptions

None

Operation

```
if ConditionPassed(cond) then
    for i = 0 to vec_len-1
        Dd[i] = abs(Dm[i])
```

Notes

Absolute value function

The function `abs(x)` means a copy of x with its sign bit forced to zero, as defined in the Appendix to the IEEE 754-1985 standard.

Flush-to-zero mode

The FZ bit of the FPSCR does not affect the operand or result of this instruction.

Vectors   When the LEN field of the FPSCR indicates scalar mode (vector length 1), `FABSD` performs just one absolute value operation, and `vec_len=1`, `Dd[0]=Dd`, and `Dm[0]=Dm`.

When the LEN field indicates a vector mode (vector length > 1), `FABSD` might perform more than one absolute value operation. *Addressing Mode 4 - Double-precision vectors (monadic)* on page C5-19 describes how `FABSD` encodes the registers it uses and how `vec_len`, `Dd[i]`, and `Dm[i]` are determined.

Signaling NaNs

To comply with the VFP architecture, `FABSD` must not generate an exception even if the value in its source register is a signaling NaN. This is a more stringent requirement than the one in the Appendix to the IEEE 754-1985 standard.

C4.1.2 FABSS

31			28	27	26	25	24	23	22	21	20	19	18	17	16	15			12	11	10	9	8	7	6	5	4	3			0
cond				1	1	1	0	1	D	1	1	0	0	0	0	Fd				1	0	1	0	1	1	M	0	Fm			
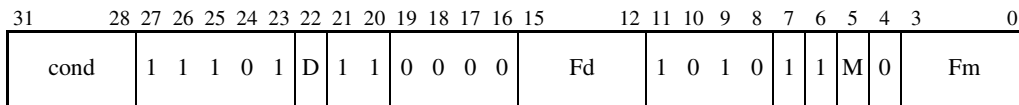
The FABSS (Floating-point Absolute Value, Single-precision) instruction writes the absolute value of a single-precision register to another single-precision register. It can also perform a vector version of this operation.

Syntax

```
FABSS{<cond>}   <Sd>, <Sm>
```

where:

<cond>   Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-7. If <cond> is omitted, the AL (always) condition is used.

<Sd>   Specifies the destination register. Its number is encoded as Fd (top 4 bits) and D (bottom bit).

<Sm>   Specifies the source register. Its number is encoded as Fm (top 4 bits) and M (bottom bit).

Architecture version

All

Exceptions

None

Operation

```
if ConditionPassed(cond) then
    for i = 0 to vec_len-1
        Sd[i] = abs(Sm[i])
```

Notes

Absolute value function

The function abs(x) means a copy of x with its sign bit forced to zero, as defined in the Appendix to the IEEE 754-1985 standard.

Flush-to-zero mode

The FZ bit of the FPSCR does not affect the operand or result of this instruction.

Vectors When the LEN field of the FPSCR indicates scalar mode (vector length 1), FABSS performs just one absolute value operation, and vec_len=1, Sd[0]=Sd, and Sm[0]=Sm.

When the LEN field indicates a vector mode (vector length > 1), FABSS might perform more than one absolute value operation. *Addressing Mode 3 - Single-precision vectors (monadic)* on page C5-14 describes how FABSS encodes the registers it uses and how vec_len, Sd[i], and Sm[i] are determined.

Signaling NaNs

To comply with the VFP architecture, FABSS must not generate an exception even if the value in its source register is a signaling NaN. This is a more stringent requirement than the one in the Appendix to the IEEE 754-1985 standard.

C4.1.3 FADDD

31		28	27	26	25	24	23	22	21	20	19		16	15		12	11	10	9	8	7	6	5	4	3		0
cond			1	1	1	0	0	0	1	1	Dn			Dd			1	0	1	1	0	0	0	0	Dm		
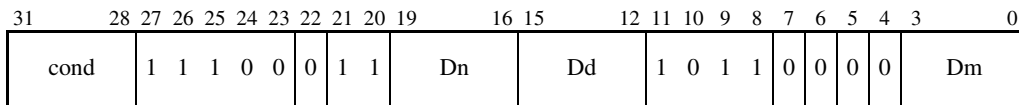
The FADDD (Floating-point Addition, Double-precision) instruction adds together two double-precision registers and writes the result to a third double-precision register. It can also perform a vector version of this operation.

Syntax

```
FADDD{<cond>}  <Dd>, <Dn>, <Dm>
```

where:

<cond>   Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-7. If <cond> is omitted, the AL (always) condition is used.

<Dd>   Specifies the destination register.

<Dn>   Specifies the register that contains the first operand for the addition.

<Dm>   Specifies the register that contains the second operand for the addition.

Architecture version

D variants only

Exceptions

Floating-point exceptions: Invalid Operation, Overflow, Inexact

Operation

```
if ConditionPassed(cond) then
    for i = 0 to vec_len-1
        Dd[i] = Dn[i] + Dm[i]
```

Notes

Vectors   When the LEN field of the FPSCR indicates scalar mode (vector length 1), FADDD performs just one addition, and vec_len=1, Dd[0]=Dd, Dn[0]=Dn, and Dm[0]=Dm.

When the LEN field indicates a vector mode (vector length > 1), FADDD might perform more than one addition. *Addressing Mode 2 - Double-precision vectors (non-monadic)* on page C5-8 describes how FADDD encodes the registers it uses and how vec_len, Dd[i], Dn[i], and Dm[i] are determined.

Rounding   The operation is a fully-rounded addition. The rounding mode is determined by the FPSCR.

C4.1.4   FADDS

31		28	27	26	25	24	23	22	21	20	19		16	15		12	11	10	9	8	7	6	5	4	3		0
cond			1	1	1	0	0	D	1	1	Fn			Fd			1	0	1	0	N	0	M	0	Fm		
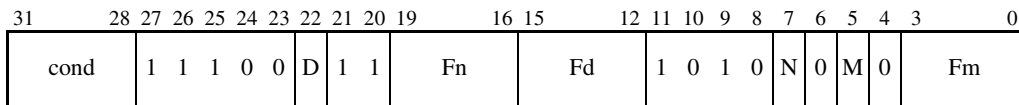
The FADDS (Floating-point Addition, Single-precision) instruction adds together two single-precision registers and writes the result to a third single-precision register. It can also perform a vector version of this operation.

Syntax

```
FADDS{<cond>}  <Sd>, <Sn>, <Sm>
```

where:

<cond>	Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-7. If <cond> is omitted, the AL (always) condition is used.
<Sd>	Specifies the destination register. Its number is encoded as Fd (top 4 bits) and D (bottom bit).
<Sn>	Specifies the register that contains the first operand for the addition. Its number is encoded as Fn (top 4 bits) and N (bottom bit).
<Sm>	Specifies the register that contains the second operand for the addition. Its number is encoded as Fm (top 4 bits) and M (bottom bit).

Architecture version

All

Exceptions

Floating-point exceptions: Invalid Operation, Overflow, Inexact

Operation

```
if ConditionPassed(cond) then
    for i = 0 to vec_len-1
        Sd[i] = Sn[i] + Sm[i]
```

Notes

Vectors   When the LEN field of the FPSCR indicates scalar mode (vector length 1), FADDS performs just one addition, and vec_len=1, Sd[0]=Sd, Sn[0]=Sn, and Sm[0]=Sm.

When the LEN field indicates a vector mode (vector length > 1), FADDS might perform more than one addition. *Addressing Mode 1 - Single-precision vectors (non-monadic)* on page C5-2 describes how FADDS encodes the registers it uses and how vec_len, Sd[i], Sn[i], and Sm[i] are determined.

Rounding   The operation is a fully-rounded addition. The rounding mode is determined by the FPSCR.

C4.1.5 **FCMPD**

31	28	27	26	25	24	23	22	21	20	19			16	15			12	11	10	9	8	7	6	5	4	3		0
cond		1	1	1	0	1	0	1	1	0	1	0	0		Dd			1	0	1	1	0	1	0	0		Dm	
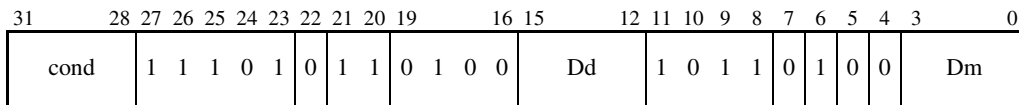
The FCMPD (Floating-point Compare, Double-precision) instruction compares two double-precision registers, writing the result to the FPSCR flags (which is normally transferred to the ARM flags by a subsequent FMSTAT instruction).

Syntax

```
FCMPD{<cond>}  <Dd>, <Dm>
```

where:

<cond>   Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-7. If <cond> is omitted, the AL (always) condition is used.

<Dd>   Specifies the register which contains the first operand for the comparison.

<Dm>   Specifies the register which contains the second operand for the comparison.

Architecture version

D variants only

Exceptions

Floating-point exceptions: Invalid Operation

Operation

```
if ConditionPassed(cond) then
    if (Dd is a signaling NaN) or (Dm is a signaling NaN) then
        raise Invalid Operation exception
    FPSCR N flag = if (Dd < Dm) then 1 else 0
    FPSCR Z flag = if (Dd == Dm) then 1 else 0
    FPSCR C flag = if (Dd < Dm) then 0 else 1
    FPSCR V flag = if (Dd and Dm compare as unordered) then 1 else 0
```

Notes

Vectors   FCMPD always specifies a scalar operation, regardless of the LEN field of the FPSCR.

NaNs   The IEEE 754 standard specifies that the result of a comparison is precisely one of <, ==, > or *unordered*. If either or both of Dd and Dm are NaNs, they are unordered, and all three of (Dd < Dm), (Dd == Dm) and (Dd > Dm) are false. This results in the FPSCR flags being set as N=0, Z=0, C=1 and V=1.

FCMPD only raises an Invalid Operation exception if one or both operands are signaling NaNs, and is suitable for testing for ==, ! =, unorderedness, and other predicates which do not raise an exception when the operands are unordered.

C4.1.6 **FCMPED**

31			28	27	26	25	24	23	22	21	20	19	18	17	16	15		12	11	10	9	8	7	6	5	4	3		0
cond				1	1	1	0	1	0	1	1	0	1	0	0	Dd			1	0	1	1	1	1	0	0	Dm		

The FCMPED (Floating-point Compare (NaN Exceptions), Double-precision) instruction compares two double-precision registers, writing the result to the FPSCR flags (which is normally transferred to the ARM flags by a subsequent FMSTAT instruction).

Syntax

```
FCMPED{<cond>}  <Dd>, <Dm>
```

where:

<cond>	Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-7. If <cond> is omitted, the AL (always) condition is used.
<Dd>	Specifies the register which contains the first operand for the comparison.
<Dm>	Specifies the register which contains the second operand for the comparison.

Architecture version

D variants only

Exceptions

Floating-point exceptions: Invalid Operation

Operation

```
if ConditionPassed(cond) then
    if (Dd is a NaN) or (Dm is a NaN) then
        raise Invalid Operation exception
    FPSCR N flag = if (Dd < Dm) then 1 else 0
    FPSCR Z flag = if (Dd == Dm) then 1 else 0
    FPSCR C flag = if (Dd < Dm) then 0 else 1
    FPSCR V flag = if (Dd and Dm compare as unordered) then 1 else 0
```

Notes

Vectors   `FCMPED` always specifies a scalar operation, regardless of the LEN field of the FPSCR.

NaNs   The IEEE 754 standard specifies that the result of a comparison is precisely one of `<`, `==`, `>` or *unordered*. If either or both of Dd and Dm are NaNs, they are unordered, and all three of (Dd < Dm), (Dd == Dm) and (Dd > Dm) are false. This results in the FPSCR flags being set as N=0, Z=0, C=1 and V=1.

   `FCMPED` raises an Invalid Operation exception if one or both operands are any type of NaN, and is suitable for testing for `<`, `<=`, `>`, `>=`, and other predicates which raise an exception when the operands are unordered.

C4.1.7   FCMPES

31	28	27	26	25	24	23	22	21	20	19	18	17	16	15	12	11	10	9	8	7	6	5	4	3	0
cond		1	1	1	0	1	D	1	1	0	1	0	0	Fd		1	0	1	0	1	1	M	0	Fm	
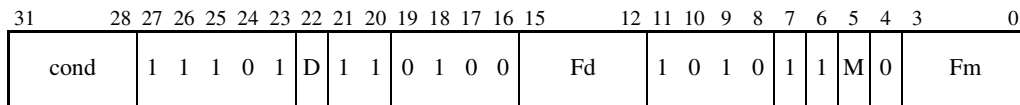
The `FCMPES` (Floating-point Compare (NaN Exceptions), Single-precision) instruction compares two single-precision registers, writing the result to the FPSCR flags (which is normally transferred to the ARM flags by a subsequent `FMSTAT` instruction).

Syntax

```
FCMPES{<cond>}  <Sd>, <Sm>
```

where:

`<cond>`	Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-7. If `<cond>` is omitted, the AL (always) condition is used.
`<Sd>`	Specifies the register which contains the first operand for the comparison. The register number is encoded as Fd (top 4 bits) and D (bottom bit).
`<Sm>`	Specifies the register which contains the second operand for the comparison. The register number is encoded as Fm (top 4 bits) and M (bottom bit).

Architecture version

All

Exceptions

Floating-point exceptions: Invalid Operation

Operation

```
if ConditionPassed(cond) then
    if (Sd is a NaN) or (Sm is a NaN) then
        raise Invalid Operation exception
    FPSCR N flag = if (Sd < Sm) then 1 else 0
    FPSCR Z flag = if (Sd == Sm) then 1 else 0
    FPSCR C flag = if (Sd < Sm) then 0 else 1
    FPSCR V flag = if (Sd and Sm compare as unordered) then 1 else 0
```

Notes

Vectors   `FCMPES` always specifies a scalar operation, regardless of the LEN field of the FPSCR.

NaNs   The IEEE 754 standard specifies that the result of a comparison is precisely one of `<`, `==`, `>` or *unordered*. If either or both of Dd and Dm are NaNs, they are unordered, and all three of (Dd < Dm), (Dd == Dm) and (Dd > Dm) are false. This results in the FPSCR flags being set as N=0, Z=0, C=1 and V=1.

   `FCMPES` raises an Invalid Operation exception if the operand is any type of NaN, and is suitable for testing for `<`, `<=`, `>`, `>=`, and other predicates which raise an exception when the operands are unordered.

C4.1.8 FCMPEZD

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

The `FCMPEZD` (Floating-point Compare (NaN Exceptions) with Zero, Double-precision) instruction compares a double-precision register with zero, writing the result to the FPSCR flags (which is normally transferred to the ARM flags by a subsequent `FMSTAT` instruction).

Syntax

```
FCMPEZD{<cond>}   <Dd>
```

where:

<cond>   Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-7. If `<cond>` is omitted, the AL (always) condition is used.

<Dd>   Specifies the register which contains the first operand for the comparison.

Architecture version

D variants only

Exceptions

Floating-point exceptions: Invalid Operation

Operation

```
if ConditionPassed(cond) then
    if (Dd is a NaN) then
        raise Invalid Operation exception
    FPSCR N flag = if (Dd < 0.0) then 1 else 0
    FPSCR Z flag = if (Dd == 0.0) then 1 else 0
    FPSCR C flag = if (Dd < 0.0) then 0 else 1
    FPSCR V flag = if (Dd is a NaN) then 1 else 0
```

Notes

Vectors   FCMPEZD always specifies a scalar operation, regardless of the LEN field of the FPSCR.

NaNs   The IEEE 754 standard specifies that the result of a comparison is precisely one of <, ==, > or *unordered*. If Dd is a NaN, it compares as unordered with zero, and all three of (Dd < 0.0), (Dd == 0.0) and (Dd > 0.0) are false. This results in the FPSCR flags being set as N=0, Z=0, C=1 and V=1.

FCMPEZD raises an Invalid Operation exception if the operand is any type of NaN, and is suitable for testing for <, <=, >, >=, and other predicates which raise an exception when the operands are unordered.

C4.1.9 FCMPEZS

31		28	27	26	25	24	23	22	21	20	19	18	17	16	15		12	11	10	9	8	7	6	5	4	3	2	1	0
cond			1	1	1	0	1	D	1	1	0	1	0	1	Fd			1	0	1	0	1	1	0	0	SBZ			

The FCMPEZS (Floating-point Compare (NaN Exceptions) with Zero, Single-precision) instruction compares a single-precision register with zero, writing the result to the FPSCR flags (which is normally transferred to the ARM flags by a subsequent FMSTAT instruction).

Syntax

```
FCMPEZS{<cond>}  <Sd>
```

where:

<cond>   Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-7. If <cond> is omitted, the AL (always) condition is used.

<Sd>   Specifies the register which contains the first operand for the comparison. The register number is encoded as Fd (top 4 bits) and D (bottom bit).

Architecture version

All

Exceptions

Floating-point exceptions: Invalid Operation

Operation

```
if ConditionPassed(cond) then
    if (Sd is a NaN) then
        raise Invalid Operation exception
    FPSCR N flag = if (Sd < 0.0) then 1 else 0
    FPSCR Z flag = if (Sd == 0.0) then 1 else 0
    FPSCR C flag = if (Sd < 0.0) then 0 else 1
    FPSCR V flag = if (Sd is a NaN) then 1 else 0
```

Notes

Vectors   FCMPEZS always specifies a scalar operation, regardless of the LEN field of the FPSCR.

NaNs   The IEEE 754 standard specifies that the result of a comparison is precisely one of <, ==, > or *unordered*. If Dd is a NaN, it compares as unordered with zero, and all three of (Dd < 0.0), (Dd == 0.0) and (Dd > 0.0) are false. This results in the FPSCR flags being set as N=0, Z=0, C=1 and V=1.

FCMPEZS raises an Invalid Operation exception if the operand is any type of NaN, and is suitable for testing for <, <=, >, >=, and other predicates which raise an exception when the operands are unordered.

C4.1.10 FCMPS

31	28	27	26	25	24	23	22	21	20	19	18	17	16	15		12	11	10	9	8	7	6	5	4	3		0
cond		1	1	1	0	1	D	1	1	0	1	0	0		Fd		1	0	1	0	0	1	M	0		Fm	
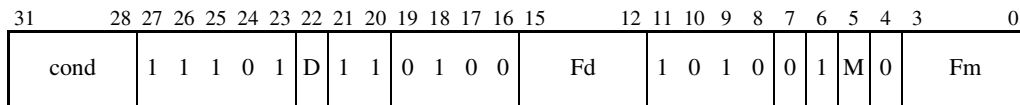
The FCMPS (Floating-point Compare, Single-precision) instruction compares two single-precision registers, writing the result to the FPSCR flags (which is normally transferred to the ARM flags by a subsequent FMSTAT instruction).

Syntax

```
FCMPS{<cond>}  <Sd>, <Sm>
```

where:

<cond>   Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-7. If <cond> is omitted, the AL (always) condition is used.

<Sd>   Specifies the register which contains the first operand for the comparison. The register number is encoded as Fd (top 4 bits) and D (bottom bit).

<Sm>   Specifies the register which contains the second operand for the comparison. The register number is encoded as Fm (top 4 bits) and M (bottom bit).

Architecture version

All

Exceptions

Floating-point exceptions: Invalid Operation

Operation

```
if ConditionPassed(cond) then
    if (Sd is a signaling NaN) or (Sm is a signaling NaN) then
        raise Invalid Operation exception
    FPSCR N flag = if (Sd < Sm) then 1 else 0
    FPSCR Z flag = if (Sd == Sm) then 1 else 0
    FPSCR C flag = if (Sd < Sm) then 0 else 1
    FPSCR V flag = if (Sd and Sm compare as unordered) then 1 else 0
```

Notes

Vectors   `FCMPS` always specifies a scalar operation, regardless of the LEN field of the FPSCR.

NaNs   The IEEE 754 standard specifies that the result of a comparison is precisely one of `<`, `==`, `>` or *unordered*. If either or both of Dd and Dm are NaNs, they are unordered, and all three of (Dd < Dm), (Dd == Dm) and (Dd > Dm) are false. This results in the FPSCR flags being set as N=0, Z=0, C=1 and V=1.

C4.1.11 FCMPZD

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

The FCMPZD (Floating-point Compare with Zero, Double-precision) instruction compares a double-precision register with zero, writing the result to the FPSCR flags (which is normally transferred to the ARM flags by a subsequent FMSTAT instruction).

Syntax

```
FCMPZD{<cond>}   <Dd>
```

where:

<cond>   Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-7. If <cond> is omitted, the AL (always) condition is used.

<Dd>   Specifies the register which contains the first operand for the comparison.

Architecture version

D variants only

Exceptions

Floating-point exceptions: Invalid Operation

Operation

```
if ConditionPassed(cond) then
    if (Dd is a signaling NaN) then
        raise Invalid Operation exception
    FPSCR N flag = if (Dd < 0.0) then 1 else 0
    FPSCR Z flag = if (Dd == 0.0) then 1 else 0
    FPSCR C flag = if (Dd < 0.0) then 0 else 1
    FPSCR V flag = if (Dd is a NaN) then 1 else 0
```

Notes

Vectors   `FCMPZD` always specifies a scalar operation, regardless of the LEN field of the FPSCR.

NaNs   The IEEE 754 standard specifies that the result of a comparison is precisely one of <, ==, >
                or *unordered*. If Dd is a NaN, it compares as unordered with zero, and all three of (Dd <
                0.0), (Dd == 0.0) and (Dd > 0.0) are false. This results in the FPSCR flags being set as N=0,
                Z=0, C=1 and V=1.

                `FCMPZD` only raises an Invalid Operation exception if the operand is a signaling NaN, and
                is suitable for testing for ==, !=, unorderedness, and other predicates which do not raise an
                exception when the operands are unordered.

C4.1.12 FCMPZS

31		28	27	26	25	24	23	22	21	20	19	18	17	16	15		12	11	10	9	8	7	6	5	4	3	2	1	0
cond			1	1	1	0	1	D	1	1	0	1	0	1	Fd			1	0	1	0	0	1	0	0		SBZ		
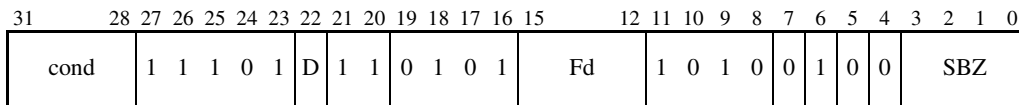
The FCMPZS (Floating-point Compare with Zero, Single-precision) instruction compares a single-precision register with zero, writing the result to the FPSCR flags (which is normally transferred to the ARM flags by a subsequent FMSTAT instruction).

Syntax

```
FCMPZS{<cond>}  <Sd>
```

where:

<cond>   Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-7. If <cond> is omitted, the AL (always) condition is used.

<Sd>   Specifies the register which contains the first operand for the comparison. The register number is encoded as Fd (top 4 bits) and D (bottom bit).

Architecture version

All

Exceptions

Floating-point exceptions: Invalid Operation

Operation

```
if ConditionPassed(cond) then
    if (Sd is a signaling NaN) then
        raise Invalid Operation exception
    FPSCR N flag = if (Sd < 0.0) then 1 else 0
    FPSCR Z flag = if (Sd == 0.0) then 1 else 0
    FPSCR C flag = if (Sd < 0.0) then 0 else 1
    FPSCR V flag = if (Sd is a NaN) then 1 else 0
```

Notes

Vectors   `FCMPZS` always specifies a scalar operation, regardless of the LEN field of the FPSCR.

NaNs   The IEEE 754 standard specifies that the result of a comparison is precisely one of <, ==, > or *unordered*. If Dd is a NaN, it compares as unordered with zero, and all three of (Dd < 0.0), (Dd == 0.0) and (Dd > 0.0) are false. This results in the FPSCR flags being set as N=0, Z=0, C=1 and V=1.

`FCMPZS` only raises an Invalid Operation exception if the operand is a signaling NaNs, and is suitable for testing for ==, ! =, unorderedness, and other predicates which do not raise an exception when the operands are unordered.

C4.1.13 FCPYD

31	28	27	26	25	24	23	22	21	20	19	18	17	16	15	12	11	10	9	8	7	6	5	4	3	0
cond		1	1	1	0	1	0	1	1	0	0	0	0	Dd		1	0	1	1	0	1	0	0	Dm	
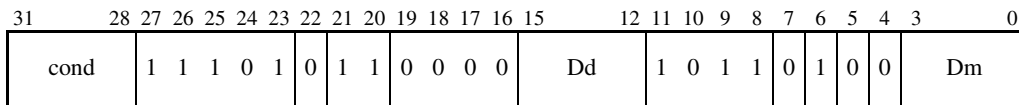
The FCPYD (Floating-point Copy, Double-precision) instruction copies one double-precision register to another double-precision register. It can also perform a vector version of this operation.

Syntax

```
FCPYD{<cond>}  <Dd>, <Dm>
```

where:

<cond>   Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-7. If <cond> is omitted, the AL (always) condition is used.

<Dd>   Specifies the destination register.

<Dm>   Specifies the source register.

Architecture version

D variants only

Exceptions

None

Operation

```
if ConditionPassed(cond) then
    for i = 0 to vec_len-1
        Dd[i] = Dm[i]
```

Notes

Vectors   When the LEN field of the FPSCR indicates scalar mode (vector length 1), FCPYD performs just one copy, and vec_len=1, Dd[0]=Dd, and Dm[0]=Dm.

When the LEN field indicates a vector mode (vector length > 1), FCPYD might perform more than one copy. *Addressing Mode 4 - Double-precision vectors (monadic)* on page C5-19 describes how FCPYD encodes the registers it uses and how vec_len, Dd[i], and Dm[i] are determined.

Flush-to-zero mode

The FZ bit of the FPSCR does not affect the operand or result of this instruction.

Signaling NaNs

To comply with the VFP architecture, FCPYD must not generate an exception even if the value in its source register is a signaling NaN. This is a more stringent requirement than the one in the IEEE 754-1985 standard.

C4.1.14 FCPYS

31	28	27	26	25	24	23	22	21	20	19	18	17	16	15	12	11	10	9	8	7	6	5	4	3	0
cond		1	1	1	0	1	D	1	1	0	0	0	0	Fd		1	0	1	0	0	1	M	0	Fm	
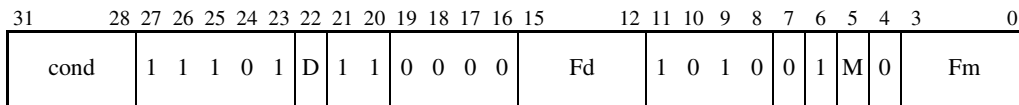
The FCPYS (Floating-point Copy, Single-precision) instruction copies one single-precision register to another single-precision register. It can also perform a vector version of this operation.

Syntax

```
FCPYS{<cond>}  <Sd>, <Sm>
```

where:

<cond>	Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-7. If <cond> is omitted, the AL (always) condition is used.
<Sd>	Specifies the destination register. The register number is encoded as Fd (top 4 bits) and D (bottom bit).
<Sm>	Specifies the source register. The register number is encoded as Fm (top 4 bits) and M (bottom bit).

Architecture version

All

Exceptions

None

Operation

```
if ConditionPassed(cond) then
    for i = 0 to vec_len-1
        Sd[i] = Sm[i]
```

Notes

Vectors   When the LEN field of the FPSCR indicates scalar mode (vector length 1), `FCPYS` performs just one copy, and `vec_len=1`, `Sd[0]=Sd`, and `Sm[0]=Sm`.

When the LEN field indicates a vector mode (vector length > 1), `FCPYD` might perform more than one copy. *Addressing Mode 3 - Single-precision vectors (monadic)* on page C5-14 describes how `FCPYS` encodes the registers it uses and how `vec_len`, `Sd[i]`, and `Sm[i]` are determined.

Flush-to-zero mode

The FZ bit of the FPSCR does not affect the operand or result of this instruction.

Signaling NaNs

To comply with the VFP architecture, `FCPYS` must not generate an exception even if the value in its source register is a signaling NaN. This is a more stringent requirement than the one in the IEEE 754-1985 standard.

C4.1.15 FCVTDS

31	28	27	26	25	24	23	22	21	20	19	18	17	16	15	12	11	10	9	8	7	6	5	4	3	0
cond		1	1	1	0	1	0	1	1	0	1	1	1	Dd		1	0	1	0	1	1	M	0	Fm	
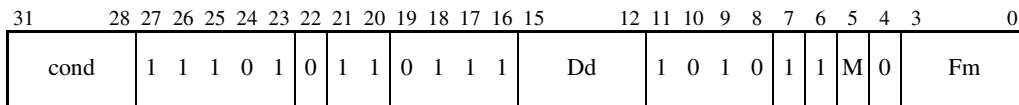
The FCVTDS (Floating-point Convert to Double-precision from Single-precision) instruction converts the value in a single-precision register to double-precision and writes the result to a double-precision register.

Syntax

```
FCVTDS{<cond>}  <Dd>, <Sm>
```

where:

<cond>   Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-7. If <cond> is omitted, the AL (always) condition is used.

<Dd>   Specifies the destination register.

<Sm>   Specifies the source register. The register number is encoded as Fm (top 4 bits) and M (bottom bit).

Architecture version

D variants only

Exceptions

Floating-point exception: Invalid Operation

Operation

```
if ConditionPassed(cond) then
    Dd = ConvertSingleToDouble(Sm)
```

Notes

Vectors   FCVTDS always specifies a scalar operation, regardless of the LEN field of the FPSCR.

C4.1.16 FCVTSD

31	28	27	26	25	24	23	22	21	20	19	18	17	16	15	12	11	10	9	8	7	6	5	4	3	0
cond		1	1	1	0	1	D	1	1	0	1	1	1		Fd	1	0	1	1	1	1	0	0		Dm
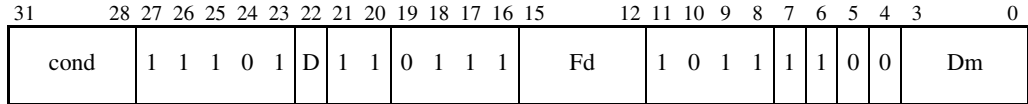
The FCVTSD (Floating-point Convert to Single-precision from Double-precision) instruction converts the value in a double-precision register to single-precision and writes the result to a single-precision register.

Syntax

```
FCVTSD{<cond>}  <Sd>, <Dm>
```

where:

<cond>   Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-7. If <cond> is omitted, the AL (always) condition is used.

<Sd>   Specifies the destination register. The register number is encoded as Fd (top 4 bits) and D (bottom bit).

<Dm>   Specifies the source register.

Architecture version

D variants only

Exceptions

Floating-point exceptions: Invalid Operation, Overflow, Underflow, Inexact

Operation

```
if ConditionPassed(cond) then
    Sd = ConvertDoubleToSingle(Dm)
```

Notes

Vectors   FCVTSD always specifies a scalar operation, regardless of the LEN field of the FPSCR.

Rounding   FCVTSD performs a fully-rounded conversion. The rounding mode is determined by the FPSCR.

C4.1.17 FDIVD

31				28	27	26	25	24	23	22	21	20	19				16	15				12	11	10	9	8	7	6	5	4	3				0
cond					1	1	1	0	1	0	0	0	Dn					Dd					1	0	1	1	0	0	0	0	Dm				
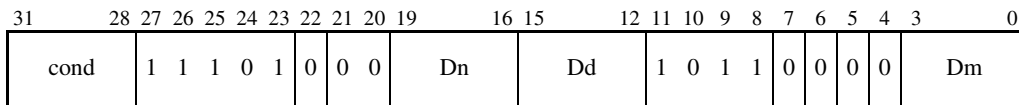
The FDIVD (Floating-point Divide, Double-precision) instruction divides one double-precision register by another double-precision register and writes the result to a third double-precision register. It can also perform a vector version of this operation.

Syntax

```
FDIVD{<cond>}  <Dd>, <Dn>, <Dm>
```

where:

<cond>   Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-7. If <cond> is omitted, the AL (always) condition is used.

<Dd>   Specifies the destination register.

<Dn>   Specifies the register that contains the first operand for the division.

<Dm>   Specifies the register that contains the second operand for the division.

Architecture version

D variants only

Exceptions

Floating-point exceptions: Invalid Operation, Division by Zero, Overflow, Underflow, Inexact

Operation

```
if ConditionPassed(cond) then
    for i = 0 to vec_len-1
        Dd[i] = Dn[i] / Dm[i]
```

Usage

Divisions take a large number of cycles on most implementations, and vector divisions take proportionately longer. This can have a major effect on performance.

If a lot of divisions by the same number are wanted, the performance can usually be improved by using one division to calculate the number's reciprocal, followed by a lot of multiplications by that reciprocal. This slightly reduces the accuracy of the calculations, since they incur two rounding errors rather than one, but this is often an acceptable trade-off.

Also see *Interrupts* on page C1-6 for a description of some implications for interrupt latency.

Notes

Vectors   When the LEN field of the FPSCR indicates scalar mode (vector length 1), `FDIVD` performs just one division, and `vec_len=1`, `Dd[0]=Dd`, `Dn[0]=Dn`, and `Dm[0]=Dm`.

When the LEN field indicates a vector mode (vector length > 1), `FDIVD` might perform more than one division. *Addressing Mode 2 - Double-precision vectors (non-monadic)* on page C5-8 describes how `FDIVD` encodes the registers it uses and how `vec_len`, `Dd[i]`, `Dn[i]`, and `Dm[i]` are determined.

Rounding   The operation is a fully-rounded division. The rounding mode is determined by the FPSCR.

C4.1.18 FDIVS

31				28	27	26	25	24	23	22	21	20	19			16	15			12	11	10	9	8	7	6	5	4	3			0
cond					1	1	1	0	1	D	0	0		Fn				Fd			1	0	1	0	N	0	M	0		Fm	
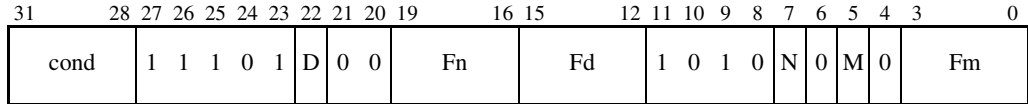
The `FDIVS` (Floating-point Divide, Single-precision) instruction divides one single-precision register by another single-precision register and writes the result to a third single-precision register. It can also perform a vector version of this operation.

Syntax

```
FDIVS{<cond>}  <Sd>, <Sn>, <Sm>
```

where:

`<cond>`	Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-7. If `<cond>` is omitted, the `AL` (always) condition is used.
`<Sd>`	Specifies the destination register. The register number is encoded as Fd (top 4 bits) and D (bottom bit).
`<Sn>`	Specifies the register that contains the first operand for the division. The register number is encoded as Fn (top 4 bits) and N (bottom bit).
`<Sm>`	Specifies the register that contains the second operand for the division. The register number is encoded as Fm (top 4 bits) and M (bottom bit).

Architecture version

All

Exceptions

Floating-point exceptions: Invalid Operation, Division by Zero, Overflow, Underflow, Inexact

Operation

```
if ConditionPassed(cond) then
    for i = 0 to vec_len-1
        Sd[i] = Sn[i] / Sm[i]
```

Usage

Divisions take a large number of cycles on most implementations, and vector divisions take proportionately longer. This can have a major effect on performance.

If a lot of divisions by the same number are wanted, the performance can usually be improved by using one division to calculate the number's reciprocal, followed by a lot of multiplications by that reciprocal. This slightly reduces the accuracy of the calculations, since they incur two rounding errors rather than one, but this is often an acceptable trade-off.

Also see *Interrupts* on page C1-6 for a description of some implications for interrupt latency.

Notes

Vectors   When the LEN field of the FPSCR indicates scalar mode (vector length 1), `FDIVS` performs just one division, and `vec_len=1`, `Sd[0]=Sd`, `Sn[0]=Sn`, and `Sm[0]=Sm`.

When the LEN field indicates a vector mode (vector length > 1), `FDIVS` might perform more than one division. The way `FDIVS` encodes the registers it uses and how `vec_len`, `Sd[i]`, `Sn[i]`, and `Sm[i]` are determined is described on *Addressing Mode 1 - Single-precision vectors (non-monadic)* on page C5-2.

Rounding   The operation is a fully-rounded division. The rounding mode is determined by the FPSCR.

C4.1.19 FLDD

31	28	27	26	25	24	23	22	21	20	19	16	15	12	11	10	9	8	7	0
cond		1	1	0	1	U	0	0	1		Rn		Dd	1	0	1	1		offset
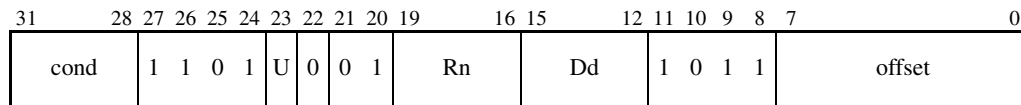
The FLDD (Floating-point Load, Double-precision) instruction loads a double-precision register from memory.

Syntax

```
FLDD{<cond>}  <Dd>, [<Rn>{, #+/-(<offset>*4)}]
```

where:

<cond>   Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-7. If <cond> is omitted, the AL (always) condition is used.

<Dd>   Specifies the destination register.

<Rn>   Specifies the register holding the base address for the transfer.

<offset>   Specifies an offset to be multiplied by 4, then added to the base address (if U == 1) or subtracted from it (if U == 0) to form the actual address of the transfer. If this offset is omitted, it defaults to +0.

Architecture version

D variants only

Exceptions

Data Abort

Operation

```
if ConditionPassed(cond) then
    if (U == 1)
        address = Rn + offset * 4
    else
        address = Rn - offset * 4
    if (big-endian)
        Dd = [Memory[address,4] << 32) OR Memory[address+4,4]
    else
        Dd = [Memory[address+4,4] << 32) OR Memory[address,4]
```

Notes

Addressing mode

This is a special case of *Addressing Mode 5 - VFP load/store multiple* on page C5-24.

Conversions   In the programmer's model, FLDD does not perform any conversion on the value transferred. Implementations are free to convert the value transferred to an internal format, provided they can recover the correct double-precision value as necessary.

C4.1.20 FLDMD

31	28	27	26	25	24	23	22	21	20	19	16	15	12	11	10	9	8	7	0
cond		1	1	0	P	U	0	W	1	Rn		Dd		1	0	1	1	offset	
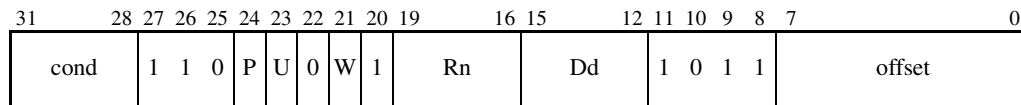
The `FLDMD` (Floating-point Load Multiple, Double-precision) instruction loads a sequence of consecutive double-precision registers from memory.

Syntax

```
FLDM<addressing_mode>D{<cond>}  <Rn>{!}, <registers>
```

where:

`<addressing_mode>`

Specifies the addressing mode, which determines the values of start_address and end_address used by the instruction. See *Addressing Mode 5 - VFP load/store multiple* on page C5-24.

`<cond>` Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-7. If `<cond>` is omitted, the `AL` (always) condition is used.

`<Rn>` Specifies the base register used by `<addressing_mode>`.

`!` Sets the W bit of the instruction to 1, specifying that the base register `<Rn>` is to be updated by the instruction. If it is omitted, the W bit of the instruction is set to 0 and the base register `<Rn>` is left unchanged. Some combinations of `<addressing_mode>` and the presence or absence of `!` are not allowed. For details, see *Addressing Mode 5 - VFP load/store multiple* on page C5-24.

`<registers>` Specifies which registers are to be loaded, as a list of consecutively numbered double-precision registers, separated by commas and surrounded by brackets. It is encoded in the instruction by setting Dd to the number of the first register in the list, and offset to twice the number of registers in the list. At least one register must be specified in the list.

For example, if `<registers>` is {D2, D3, D4}, the Dd field of the instruction is 2 and the offset field is 6.

Architecture version

D variants only

Exceptions

Data Abort

Operation

```
if ConditionPassed(cond) then
    address = start_address
    for i = 0 to (offset-2)/2
        /* d is the number of register Dd;                */
        /* D(n) is the double-precision register numbered n */
        if (big-endian)
            D(d+i) = (Memory[address,] << 32) OR Memory[address+4,4]
        else
            D(d+i) = (Memory[address+4,4] << 32) OR Memory[address,4]
        address = address + 8
    assert end_address = address - 4
```

Notes

Encoding If P=1 and W=0, the instruction is an FLDD instruction instead. Otherwise, if offset is odd, the instruction is an FLDMX instruction instead.

Vectors The FLDMD instruction is unaffected by the LEN and STRIDE fields of the FPSCR, and does not wrap around at bank boundaries in the way that vector operands to data-processing instructions do. Registers are loaded in simple increasing order of register number.

Invalid register lists

If Dd and offset do not specify a valid register list, the instruction is UNPREDICTABLE. This happens in two cases:

- if offset == 0, that is, if an attempt is made to transfer no registers

- if d + offset/2 > 16, that is, if an attempt is made to transfer another register after D15.

Conversions In the programmer's model, FLDMD does not perform any conversion on the value transferred. Implementations are free to convert the value transferred to an internal format, provided they can recover the correct double-precision value as necessary.

C4.1.21 FLDMS

31	28	27 26 25	24	23	22	21	20	19	16	15	12	11 10 9 8	7	0
cond		1 1 0	P	U	D	W	1	Rn		Fd		1 0 1 0	offset	
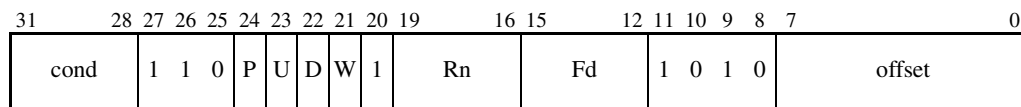
The `FLDMS` (Floating-point Load Multiple, Single-precision) instruction loads a sequence of consecutive single-precision registers from memory.

Syntax

```
FLDM<addressing_mode>S{<cond>}  <Rn>{!}, <registers>
```

where:

`<addressing_mode>`

Specifies the addressing mode, which determines the values of start_address and end_address used by the instruction. See *Addressing Mode 5 - VFP load/store multiple* on page C5-24 for details.

`<cond>`   Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-7. If `<cond>` is omitted, the `AL` (always) condition is used.

`<Rn>`   Specifies the base register used by `<addressing_mode>`.

`!`   Sets the W bit of the instruction to 1, specifying that the base register `<Rn>` is to be updated by the instruction. If it is omitted, the W bit of the instruction is set to 0 and the base register `<Rn>` is left unchanged. Some combinations of `<addressing_mode>` and the presence or absence of `!` are not allowed. For details, see *Addressing Mode 5 - VFP load/store multiple* on page C5-24.

`<registers>`   Specifies which registers are to be loaded, as a list of consecutively numbered single-precision registers, separated by commas and surrounded by brackets. If d is the number of the first register in the list, the list is encoded in the instruction by setting Fd and D to the top 4 bits and the bottom bit respectively of d, and `offset` to the number of registers in the list. At least one register must be specified in the list.

For example, if `<registers>` is {S5,S6,S7}, the Fd field of the instruction is 0b0010, the D bit is 1 and the offset field is 3.

Architecture version

All

Exceptions

Data Abort

Operation

```
if ConditionPassed(cond) then
    address = start_address
    for i = 0 to offset-1
        /* d is as defined for <registers> above;          */
        /* S(n) is the single-precision register numbered n */
        S(d+i) = Memory[address,4]
        address = address + 4
    assert end_address = address - 4
```

Notes

Encoding   If P=1 and W=0, the instruction is an FLDS instruction instead.

Vectors   The FLDMS instruction is unaffected by the LEN and STRIDE fields of the FPSCR, and does not wrap around at bank boundaries in the way that vector operands to data-processing instructions do. Registers are loaded in simple increasing order of register number.

Invalid register lists

If Fd, D and offset do not specify a valid register list, the instruction is UNPREDICTABLE. This happens in two cases:

- if offset == 0, that is, if an attempt is made to transfer no registers

- if d + offset > 32, that is, if an attempt is made to transfer another register after S31.

Conversions   In the programmer's model, FLDMS does not perform any conversion on the values transferred. The memory words can hold either integers or single-precision floating-point numbers. Most VFP arithmetic instructions treat the loaded values as single-precision floating-point numbers. If they are integers, they need to be converted using the integer-to-floating-point conversion instructions before such arithmetic instructions can yield sensible results. Implementations are free to convert the values transferred to an internal format, provided they can recover either the correct single-precision value or the correct integer value for each one (depending on how the registers are subsequently used).

C4.1.22 FLDMX

31	28	27	26	25	24	23	22	21	20	19	16	15	12	11	10	9	8	7	0
cond		1	1	0	P	U	0	W	1	Rn		Dd		1	0	1	1	offset	
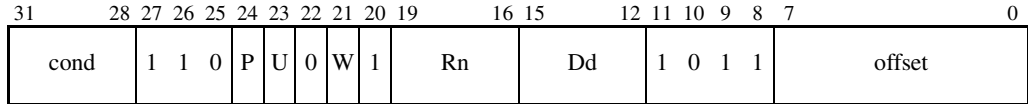
The FLDMX (Floating-point Load Multiple, Unknown precision) instruction loads a sequence of consecutive double-precision registers from memory, using the same IMPLEMENTATION DEFINED format as the FSTMX instruction. This allows the registers to be reloaded correctly with integers, single-precision values or double-precision values.

Syntax

```
FLDM<addressing_mode>X{<cond>}  <Rn>{!}, <registers>
```

where:

<addressing_mode>

> Specifies the addressing mode, which determines the values of start_address and end_address used by the instruction. See *Addressing Mode 5 - VFP load/store multiple* on page C5-24 for details.

<cond>   Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-7. If <cond> is omitted, the AL (always) condition is used.

<Rn>   Specifies the base register used by <addressing_mode>.

!   Sets the W bit of the instruction to 1, specifying that the base register <Rn> is to be updated by the instruction. If it is omitted, the W bit of the instruction is set to 0 and the base register <Rn> is left unchanged. Some combinations of <addressing_mode> and the presence or absence of ! are not allowed. For details, see *Addressing Mode 5 - VFP load/store multiple* on page C5-24.

<registers>   Specifies which registers are to be loaded, as a list of consecutively numbered double-precision registers, separated by commas and surrounded by brackets. It is encoded in the instruction by setting Dd to the number of the first register in the list, and offset to twice the number of registers in the list, plus 1. At least one register must be specified in the list.

> For example, if <registers> is {D2, D3, D4}, the Dd field of the instruction is 2 and the offset field is 7.

Architecture version

All

Exceptions

Data Abort

Operation

```
if ConditionPassed(cond) then
    /* d is the number of register Dd;                   */
    /* D(n) is the double-precision register numbered n */
    Load registers D(d) to D(d+(offset-3)/2) from memory words
      Memory[start_address,4] through to Memory[end_address-4,4]
```

Usage

`FLDMX` is used to reload VFP register values from memory when `FSTMX` was previously used to store them. Typical cases in which it is used are:

- in procedure exit sequences when a callee-save procedure-calling standard is being used
- in process swap code.

Notes

Encoding   If P=1 and W=0, the instruction is an `FLDD` instruction instead. Otherwise, if `offset` is even, the instruction is an `FLDMD` instruction instead.

Vectors   The `FLDMX` instruction is unaffected by the LEN and STRIDE fields of the FPSCR, and does not wrap around at bank boundaries in the way that vector operands to data-processing instructions do. Registers are loaded in simple increasing order of register number.

Invalid register lists

If `Dd` and `offset` do not specify a valid register list, the instruction is UNPREDICTABLE. This happens in two cases:

- if `offset == 0`, that is, if an attempt is made to transfer no registers
- if `d + (offset-1)/2 > 16`, that is, if an attempt is made to transfer another register after D15.

Memory format

The memory format used is IMPLEMENTATION DEFINED. The only requirements on it are:

- `FLDMX` must reload the register values stored by a matching `FSTMX` instruction correctly, regardless of whether they originally contained integers, single-precision values or double-precision values. For an `FLDMX` instruction and an `FSTMX` instruction to count as matching, they must specify identical register lists and must generate the same `start_address` and `end_address` when executed.

- It must not use more than the allowed number of memory words, that is, $2N+1$ memory words for storing N double-precision registers.

Non implementation-specific code must not do anything with `FLDMX` other than reload the results of a matching `FSTMX` instruction.

C4.1.23 FLDS

31	28	27	26	25	24	23	22	21	20	19	16	15	12	11	10	9	8	7	0
cond		1	1	0	1	U	D	0	1	Rn		Fd		1	0	1	0	offset	
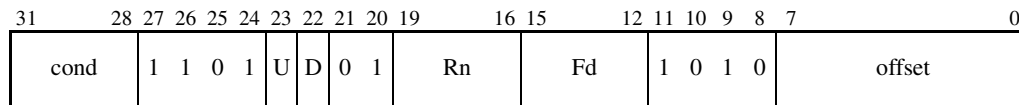
The FLDS (Floating-point Load, Single-precision) instruction loads a single-precision register from memory.

Syntax

```
FLDS{<cond>}  <Sd>, [<Rn>{, #+/-(<offset>*4)}]
```

where:

<cond>   Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-7. If <cond> is omitted, the AL (always) condition is used.

<Sd>   Specifies the destination register. Its number is encoded as Fd (top 4 bits) and D (bottom bit).

<Rn>   Specifies the register holding the base address for the transfer.

<offset>   Specifies an offset to be multiplied by 4, then added to the base address (if U == 1) or subtracted from it (if U == 0) to form the actual address of the transfer. If this offset is omitted, it defaults to +0.

Architecture version

All

Exceptions

Data Abort

Operation

```
if ConditionPassed(cond) then
    if (U == 1)
        address = Rn + offset * 4
    else
        address = Rn - offset * 4
    Sd = Memory[address,4]
```

Notes

Addressing mode

This is a special case of *Addressing Mode 5 - VFP load/store multiple* on page C5-24.

Conversions   In the programmer's model, FLDS does not perform any conversion on the value transferred. The memory word can hold either an integer or a single-precision floating-point number. Most VFP arithmetic instructions treat the Sd value as a single-precision floating-point number. If it is an integer, one of the integer-to-floating-point conversion instructions must be executed before such arithmetic instructions can yield sensible results. Implementations are free to convert the value transferred to an internal format, provided they can recover either the correct single-precision value or the correct integer value (depending on how Sd is subsequently used).

C4.1.24 FMACD

31	28	27	26	25	24	23	22	21	20	19	16	15	12	11	10	9	8	7	6	5	4	3	0
cond		1	1	1	0	0	0	0	0	Dn		Dd		1	0	1	1	0	0	0	0	Dm	
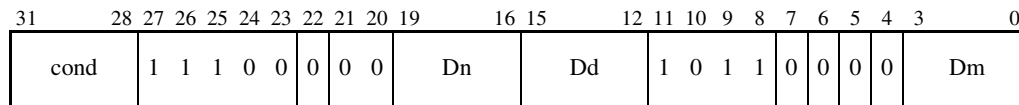
The `FMACD` (Floating-point Multiply and Accumulate, Double-precision) instruction multiplies together two double-precision registers, adds a third double-precision register to the product and writes the result to the third register. It can also perform a vector version of this operation.

Syntax

```
FMACD{<cond>} <Dd>, <Dn>, <Dm>
```

where:

<cond>   Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-7. If <cond> is omitted, the AL (always) condition is used.

<Dd>   Specifies the destination register, which is also used as the second operand for the addition.

<Dn>   Specifies the register that contains the first operand for the multiplication.

<Dm>   Specifies the register that contains the second operand for the multiplication.

Architecture version

D variants only

Exceptions

Floating-point exceptions: Invalid Operation, Overflow, Underflow, Inexact

Operation

```
if ConditionPassed(cond) then
    for i = 0 to vec_len-1
        Dd[i] = Dn[i] * Dm[i] + Dd[i]
```

Notes

Vectors   When the LEN field of the FPSCR indicates scalar mode (vector length 1), FMACD performs just one multiply-add operation, and `vec_len=1`, `Dd[0]=Dd`, `Dn[0]=Dn`, and `Dm[0]=Dm`.

When the LEN field indicates a vector mode (vector length > 1), FMACD might perform more than one multiply-add operation. *Addressing Mode 2 - Double-precision vectors (non-monadic)* on page C5-8 describes how FMACD encodes the registers it uses and how `vec_len,` `Dd[i],Dn[i]` and `Dm[i]` are determined.

Rounding   The operation is a fully-rounded multiplication followed by a fully-rounded addition. The rounding mode is determined by the FPSCR.

C4.1.25 FMACS

31		28	27	26	25	24	23	22	21	20	19			16	15			12	11	10	9	8	7	6	5	4	3			0
cond			1	1	1	0	0	D	0	0	Fn				Fd				1	0	1	0	N	0	M	0	Fm			
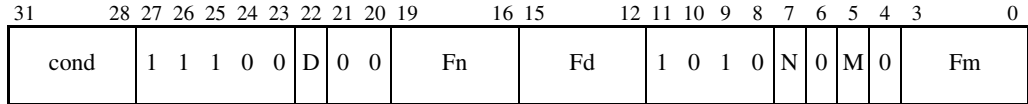
The `FMACS` (Floating-point Multiply and Accumulate, Single-precision) instruction multiplies together two single-precision registers, adds a third single-precision register to the product and writes the result to the third register. It can also perform a vector version of this operation.

Syntax

```
FMACS{<cond>} <Sd>, <Sn>, <Sm>
```

where:

<cond>   Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-7. If `<cond>` is omitted, the `AL` (always) condition is used.

<Sd>   Specifies the destination register, which is also used as the second operand for the addition. The register number is encoded as Fd (top 4 bits) and D (bottom bit).

<Sn>   Specifies the register that contains the first operand for the multiplication. The register number is encoded as Fn (top 4 bits) and N (bottom bit).

<Sm>   Specifies the register that contains the second operand for the multiplication. The register number is encoded as Fm (top 4 bits) and M (bottom bit).

Architecture version

All

Exceptions

Floating-point exceptions: Invalid Operation, Overflow, Underflow, Inexact

Operation

```
if ConditionPassed(cond) then
    for i = 0 to vec_len-1
        Sd[i] = Sn[i] * Sm[i] + Sd[i]
```

Notes

Vectors When the LEN field of the FPSCR indicates scalar mode (vector length 1), `FMACS` performs just one multiply-add operation, and `vec_len=1`, `Sd[0]=Sd`, `Sn[0]=Sn`, and `Sm[0]=Sm`.

When the LEN field indicates a vector mode (vector length > 1), `FMACS` might perform more than one multiply-add operation. *Addressing Mode 1 - Single-precision vectors (non-monadic)* on page C5-2 shows how `FMACS` encodes registers and determines `vec_len`, `Sd[i]`, `Sn[i]` and `Sm[i]`.

Rounding The operation is a fully-rounded multiplication followed by a fully-rounded addition. The rounding mode is determined by the FPSCR.

C4.1.26 FMDHR

31	28	27 26 25 24 23 22 21 20	19 16	15 12	11 10 9 8	7	6 5 4	3 0
cond		1 1 1 0 0 0 1 0	Dn	Rd	1 0 1 1	0	SBZ 1	SBZ
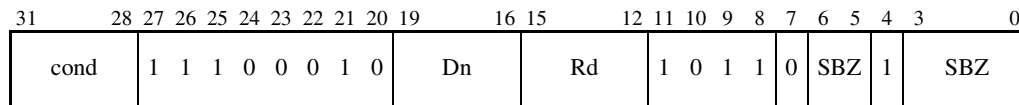
The FMDHR (Floating-point Move to Double-precision High from Register) instruction transfers the contents of the ARM register Rd to the upper half of the double-precision register Dn. It is used in conjunction with FMDLR to transfer double-precision values between ARM registers and floating-point registers.

Syntax

```
FMDHR{<cond>}   <Dn>, <Rd>
```

where:

<cond>   Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-7. If <cond> is omitted, the AL (always) condition is used.

<Dn>   Specifies the destination register.

<Rd>   Specifies the source ARM register.

Architecture version

D variants only

Exceptions

None

Operation

```
if ConditionPassed(cond) then
    Dn[63:32] = Rd
```

Notes

Use with FMDLR

FMDHR must be used in conjunction with an FMDLR instruction specifying the same destination register. Between these two instructions, the value of Dn is UNPREDICTABLE for all purposes except:

- the execution of the second instruction must result in Dn containing the double-precision number transferred by the two instructions

- if Dn is saved to memory by an FSTMX instruction and subsequently reloaded by a correctly matching FLDMX instruction, the final value of Dn must be functionally equivalent to its original value.

Conversions   In the programmer's model, the combination of FMDHR and FMDLR does not perform any conversion. Implementations are free to convert the value transferred to an internal format, provided they can recover the correct double-precision value when both the FMDHR and the FMDLR instructions have been executed.

C4.1.27 FMDLR

31	28	27	26	25	24	23	22	21	20	19	16	15	12	11	10	9	8	7	6	5	4	3	0
cond		1	1	1	0	0	0	0	0	Dn		Rd		1	0	1	1	0	SBZ		1	SBZ	

The FMDLR (Floating-point Move to Double-precision Low from Register) instruction transfers the contents of the ARM register Rd to the lower half of the double-precision register Dn. Used with FMDHR, it transfers double-precision values between ARM registers and floating-point registers.

Syntax

FMDLR{<cond>} <Dn>, <Rd>

where:

<cond>   Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-7. If <cond> is omitted, the AL (always) condition is used.

<Dn>   Specifies the destination register.

<Rd>   Specifies the source ARM register.

Architecture version

D variants only

Exceptions

None

Operation

```
if ConditionPassed(cond) then
    Dn[31:0] = Rd
```

Notes

Use with FMDHR

FMDLR must be used in conjunction with an FMDHR instruction specifying the same destination register. Between these two instructions, the value of Dn is UNPREDICTABLE for all purposes except:

- the execution of the second instruction must result in Dn containing the double-precision number transferred by the two instructions

- if Dn is saved to memory by an FSTMX instruction and subsequently reloaded by a correctly matching FLDMX instruction, the final value of Dn must be functionally equivalent to its original value.

Conversions In the programmer's model, the combination of FMDHR and FMDLR does not perform any conversion. Implementations are free to convert the value transferred to an internal format, provided they can recover the correct double-precision value when both the FMDHR and the FMDLR instructions have been executed.

C4.1.28 FMDRR

31	28	27 26 25 24 23 22 21 20	19 16	15 12	11 10 9 8	7	6	5	4	3 0
cond		1 1 0 0 0 1 0 0	Rn	Rd	1 0 1 1	0	R	0	1	Dm
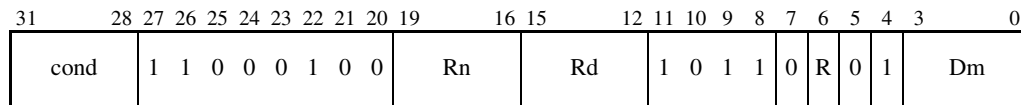
The `FMDRR` (Floating-point Move to Double-precision Register from two Registers) instruction transfers the contents of two ARM registers to a double-precision VFP register.

Syntax

```
FMDRR{<cond>}  <Dm>, <Rd>, <Rn>
```

where:

`<cond>`	Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-7. If `<cond>` is omitted, the AL (always) condition is used.
`<Dm>`	Specifies the destination double-precision VFP register.
`<Rd>`	Specifies the source ARM register for the lower half of the 64-bit operand.
`<Rn>`	Specifies the source ARM register for the upper half of the 64-bit operand.

Architecture version

D variants only, VFPv2 and above.

Exceptions

None

Operation

```
if ConditionPassed(cond) then
    Dm[63:32] = Rn
    Dm[31:0]  = Rd
```

Notes

Conversions   In the programmer's model, `FMDRR` does not perform any conversion of the value transferred. Arithmetic instructions on either of the ARM registers treat the contents as an integer. Most VFP instructions treat the Dm value as a double-precision floating-point number.

C4.1.29 FMRDH

31	28 27 26 25 24 23 22 21 20	19	16 15	12	11 10 9 8	7	6 5	4	3	0
cond	1 1 1 0 0 0 1 1	Dn	Rd		1 0 1 1	0	SBZ	1	SBZ	

The FMRDH (Floating-point Move to Register from Double-precision High) instruction transfers the upper half of the contents of the double-precision register Dn to the ARM register Rd. It is used in conjunction with FMRDL to transfer double-precision values between ARM registers and floating-point registers.

Syntax

```
FMRDH{<cond>}  <Rd>, <Dn>
```

where:

<cond>	Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-7. If <cond> is omitted, the AL (always) condition is used.
<Rd>	Specifies the destination ARM register.
<Dn>	Specifies the source register.

Architecture version

D variants only

Exceptions

None

Operation

```
if ConditionPassed(cond) then
    Rd = Dn[63:32]
```

Notes

Conversions If an implementation uses an internal format for double-precision values, it must convert that format back to the external double-precision format. Otherwise, no conversion is required.

C4.1.30 FMRDL

31	28	27	26	25	24	23	22	21	20	19	16	15	12	11	10	9	8	7	6	5	4	3	0
cond		1	1	1	0	0	0	0	1	Dn		Rd		1	0	1	1	0	SBZ		1	SBZ	

The FMRDL (Floating-point Move to Register from Double-precision Low) instruction transfers the lower half of the contents of the double-precision register Dn to the ARM register Rd. It is used in conjunction with FMRDH to transfer double-precision values between ARM registers and floating-point registers.

Syntax

```
FMRDL{<cond>}  <Rd>, <Dn>
```

where:

<cond>   Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-7. If <cond> is omitted, the AL (always) condition is used.

<Rd>   Specifies the destination ARM register.

<Dn>   Specifies the source register.

Architecture version

D variants only

Exceptions

None

Operation

```
if ConditionPassed(cond) then
    Rd = Dn[31:0]
```

Notes

Conversions   If an implementation uses an internal format for double-precision values, it must convert that format back to the external double-precision format. Otherwise, no conversion is required.

C4.1.31 FMRRD

31	28	27 26 25 24 23 22 21 20	19	16	15	12	11 10 9 8	7	6	5	4	3	0
cond		1 1 0 0 0 1 0 1	Rn		Rd		1 0 1 1	0	R	0	1	Dm	
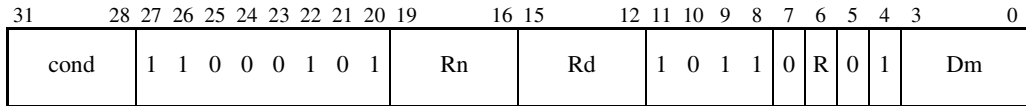
The FMRRD (Floating-point Move to two Registers from Double-precision Register) instruction transfers the contents of a double-precision VFP register to two ARM registers.

Syntax

```
FMRRD{<cond>}  <Rd, <Rn>, <Dm>
```

where:

<cond>	Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-7. If <cond> is omitted, the AL (always) condition is used.
<Rd>	Specifies the destination ARM register for the lower half of the 64-bit operand.
<Rn>	Specifies the destination ARM register for the upper half of the 64-bit operand.
<Dm>	Specifies the source double-precision VFP register.

Architecture version

D variants only, VFPv2 and above.

Exceptions

None

Operation

```
if ConditionPassed(cond) then
    Rn = Dm[63:32]
    Rd = Dm[31:0]
```

Notes

Use of R15 If R15 is specified for Rd or Rn, the results are UNPREDICTABLE.

Conversions In the programmer's model, FMRRD does not perform any conversion of the value transferred. Arithmetic instructions on either of the ARM registers treat the contents as an integer. Most VFP instructions treat the Dm value as a double-precision floating-point number.

C4.1.32 FMRRS

31 28	27 26 25 24 23 22 21 20	19 16	15 12	11 10 9 8	7	6	5	4	3 0
cond	1 1 0 0 0 1 0 1	Rn	Rd	1 0 1 0	0	R	M	1	Sm
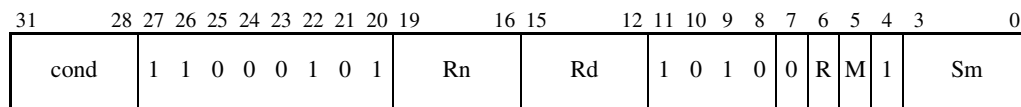
The FMRRS (Floating-point Move to two Registers from two Single-precision Registers) instruction transfers the contents of two single-precision VFP registers to two ARM registers.

Syntax

```
FMRRS{<cond>}  <Rd, <Rn>, {<Sm, Sn>}
```

where:

<cond>   Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-7. If <cond> is omitted, the AL (always) condition is used.

<Rd>   Specifies the destination ARM register for Sm.

<Rn>   Specifies the destination ARM register for Sn.

<Sm>   Specifies the first source single-precision VFP register. This is encoded in the instruction by setting Sm and M to the top 4 bits and the bottom bit respectively of m.

<Sn>   Specifies the second source single-precision VFP register. Sn must be Sm+1.

Architecture version

D variants only, VFPv2 and above.

Exceptions

None

Operation

```
if ConditionPassed(cond) then
    Rn = Sn
    Rd = Sm
```

Notes

Use of R15   If R15 is specified for `Rd` or `Rn`, the results are UNPREDICTABLE.

Conversions   In the programmer's model, `FMRRS` does not perform any conversion of the value transferred. Arithmetic instructions on either of the ARM registers treat the contents as an integer. Most VFP instructions treat the Sm and Sn values as single-precision floating-point numbers.

Invalid register lists

If `S31` is specified as `Sm` the, results are UNPREDICTABLE. If the register pair is not consecutive, an error is reported by the assembler.

C4.1.33 FMRS

31	28	27 26 25 24 23 22 21 20	19 16	15 12	11 10 9 8	7	6 5	4	3 0
cond		1 1 1 0 0 0 0 1	Fn	Rd	1 0 1 0	N	SBZ	1	SBZ

The FMRS (Floating-point Move to Register from Single-precision) instruction transfers the contents of the single-precision register Fn to the ARM register Rd. The value transferred can be an integer (typically generated by a FTOSID, FTOSIS, FTOUID or FTOUIS instruction) or a single-precision floating-point number (typically generated by other arithmetic instructions).

Syntax

```
FMRS{<cond>}  <Rd>, <Sn>
```

where:

<cond>   Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-7. If <cond> is omitted, the AL (always) condition is used.

<Rd>   Specifies the destination ARM register.

<Sn>   Specifies the source register. Its number is encoded as Fn (top 4 bits) and N (bottom bit).

Architecture version

All

Exceptions

None

Operation

```
if ConditionPassed(cond) then
    Rd = Sn
```

Notes

Conversions   In the programmer's model, FMRS does not perform any conversion on the value transferred. Both the source register Sn and the destination register Rd can contain either an integer or a single-precision floating-point number. Arithmetic instructions on the ARM treat the Rd value as an integer, whereas most arithmetic instructions on the VFP coprocessor treat the Sn value as a single-precision floating-point number. One of the floating-point-to-integer conversion instructions must be executed before the FMRS instruction if they are to agree on the number being represented.

Implementations are free to hold the value in Sn in an internal format, provided that FMRS converts it to external format and this conversion recovers the correct data, regardless of whether the register contains a single-precision floating-point number or an integer.

C4.1.34 FMRX

31	28	27	26	25	24	23	22	21	20	19	16	15	12	11	10	9	8	7	6	5	4	3	0
cond		1	1	1	0	1	1	1	1	reg		Rd		1	0	1	0	0	SBZ		1	SBZ	
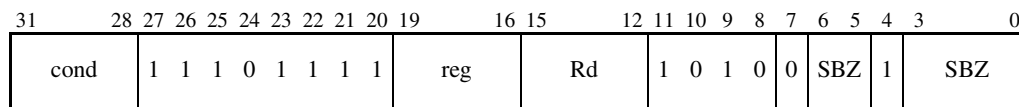
The FMRX (Floating-point Move to Register from System Register) instruction transfers the contents of one of the VFP system registers to the ARM register Rd.

Syntax

```
FMRX{<cond>}  <Rd>, <reg>
```

where:

<cond>   Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-7. If <cond> is omitted, the AL (always) condition is used.

<Rd>   Specifies the destination ARM register.

<reg>   Specifies the source system register as follows:

```
<reg> = 0b0000: FPSID
<reg> = 0b0001: FPSCR
<reg> = 0b1000: FPEXC
```

Other values of <reg> can be used by individual VFP implementations for IMPLEMENTATION DEFINED purposes. Typically, they are used to transfer data from a hardware coprocessor to the support code for that coprocessor.

All other code must treat such values of <reg> as UNPREDICTABLE.

Architecture version

All

Exceptions

None

Operation

```
if ConditionPassed(cond) then
    Rd = reg
```

Notes

Serialization   FMRX is a serializing instruction. See *System register transfer instructions* on page C3-20 for details of what this means.

Exception processing

> After serialization, if the VFP system contains a hardware coprocessor, that coprocessor might have a pending exception to process. Whether the FMRX instruction triggers the processing of such an instruction depends on which system register is being transferred, as described in the following notes. If exception processing is triggered, this causes the FMRX instruction to take the ARM's Undefined Instruction exception.

Reading FPSID

> An FMRX instruction with source FPSID can be executed in any ARM processor mode. After serialization, it writes the value of the FPSID to Rd, and does not trigger exception processing.

Reading FPSCR

> An FMRX instruction with source FPSCR can be executed in any ARM processor mode. After serialization, exception processing is triggered if necessary. Otherwise, the value of the FPSCR is written to Rd.

Reading FPEXC

> An FMRX instruction with source FPEXC can only be executed in privileged ARM processor modes. An attempt to execute it in User mode causes the ARM's Undefined Instruction exception to be taken.

> After serialization, it writes the value of FPEXC to Rd, and does not trigger exception processing. Because all but bits[31:30] of FPEXC is IMPLEMENTATION DEFINED, non implementation-specific code must only rely on bits[31:0] of the value written to Rd.

C4.1.35 FMSCD

31		28	27	26	25	24	23	22	21	20	19		16	15		12	11	10	9	8	7	6	5	4	3		0
cond			1	1	1	0	0	0	0	1		Dn			Dd		1	0	1	1	0	0	0	0		Dm	
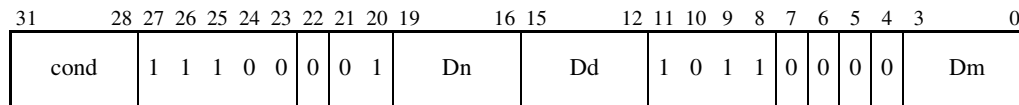
The FMSCD (Floating-point Multiply and Subtract, Double-precision) instruction multiplies together two double-precision registers, subtracts a third double-precision register from the product and writes the result to the third register. It can also perform a vector version of this operation.

Syntax

```
FMSCD{<cond>}  <Dd>, <Dn>, <Dm>
```

where:

<cond>   Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-7. If <cond> is omitted, the AL (always) condition is used.

<Dd>   Specifies the destination register, which is also the second operand for the subtraction.

<Dn>   Specifies the register that contains the first operand for the multiplication.

<Dm>   Specifies the register that contains the second operand for the multiplication.

Architecture version
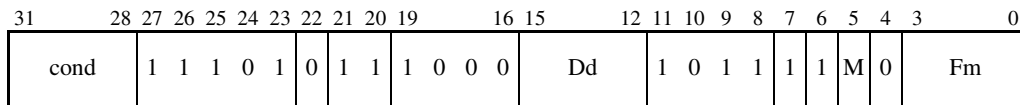
D variants only

Exceptions

Floating-point exceptions: Invalid Operation, Overflow, Underflow, Inexact

Operation

```
if ConditionPassed(cond) then
    for i = 0 to vec_len-1
        Dd[i] = Dn[i] * Dm[i] - Dd[i]
```

Notes

Vectors   When the LEN field of the FPSCR indicates scalar mode (vector length 1), FMSCD performs just one multiply-subtract operation, and vec_len=1, Dd[0]=Dd, Dn[0]=Dn, and Dm[0]=Dm.

When the LEN field indicates a vector mode (vector length > 1), FMSCD might perform more than one multiply-subtract operation. *Addressing Mode 2 - Double-precision vectors (non-monadic)* on page C5-8 describes how FMSCD encodes the registers it uses and how vec_len, Dd[i], Dn[i], and Dm[i] are determined.

Rounding   The operation is a fully-rounded multiplication followed by a fully-rounded subtraction. The rounding mode is determined by the FPSCR.

C4.1.36 FMSCS

31		28	27	26	25	24	23	22	21	20	19		16	15		12	11	10	9	8	7	6	5	4	3		0
cond			1	1	1	0	0	D	0	1	Fn			Fd			1	0	1	0	N	0	M	0	Fm		
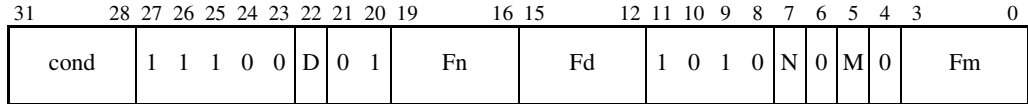
The `FMSCS` (Floating-point Multiply and Subtract, Single-precision) instruction multiplies together two single-precision registers, subtracts a third single-precision register from the product and writes the result to the third register. It can also perform a vector version of this operation.

Syntax

```
FMSCS{<cond>}  <Sd>, <Sn>, <Sm>
```

where:

<cond>	Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-7. If <cond> is omitted, the AL (always) condition is used.
<Sd>	Specifies the destination register, which is also used as the second operand for the subtraction. The register number is encoded as Fd (top 4 bits) and D (bottom bit).
<Sn>	Specifies the register that contains the first operand for the multiplication. The register number is encoded as Fn (top 4 bits) and N (bottom bit).
<Sm>	Specifies the register that contains the second operand for the multiplication. The register number is encoded as Fm (top 4 bits) and M (bottom bit).

Architecture version

All

Exceptions

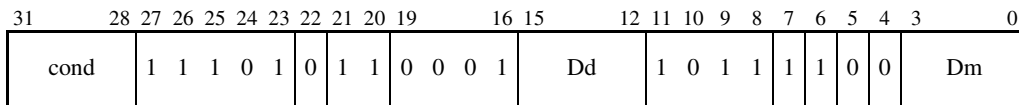Floating-point exceptions: Invalid Operation, Overflow, Underflow, Inexact

Operation

```
if ConditionPassed(cond) then
    for i = 0 to vec_len-1
        Sd[i] = Sn[i] * Sm[i] - Sd[i]
```

Notes

Vectors   When the LEN field of the FPSCR indicates scalar mode (vector length 1), `FMSCS` performs just one multiply-subtract operation, where `vec_len=1`, `Sd[0]=Sd`, `Sn[0]=Sn`, and `Sm[0]=Sm`.

When the LEN field indicates a vector mode (vector length > 1), `FMSCS` might perform more than one multiply-subtract operation. *Addressing Mode 1 - Single-precision vectors (non-monadic)* on page C5-2 shows how `FMSCS` encodes registers and determines `vec_len`, `Sd[i]`, `Sn[i]`, and `Sm[i]`.

Rounding   The operation is a fully-rounded multiplication followed by a fully-rounded subtraction. The rounding mode is determined by the FPSCR.

C4.1.37 FMSR

31	28	27	26	25	24	23	22	21	20	19	16	15	12	11	10	9	8	7	6	5	4	3	0
cond		1	1	1	0	0	0	0	0	Fn		Rd		1	0	1	0	N	SBZ		1	SBZ	

The FMSR (Floating-point Move to Single-precision from Register) instruction transfers the contents of the ARM register Rd to the single-precision register Fn. The value transferred can subsequently be treated either as an integer (if used as the source register of a FSITOD, FSITOS, FUITOD or FUITOS instruction) or as a single-precision floating-point number (if used by other arithmetic instructions).

Syntax

```
FMSR{<cond>}  <Sn>, <Rd>
```

where:

<cond>	Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-7. If <cond> is omitted, the AL (always) condition is used.
<Sn>	Is the destination register. Its number is encoded as Fn (top 4 bits) and N (bottom bit).
<Rd>	Is the source ARM register.

Architecture version

All

Exceptions

None

Operation

```
if ConditionPassed(cond) then
    Sn = Rd
```

Notes

Conversions In the programmer's model, `FMSR` does not perform any conversion on the value transferred. Both the source register `Rd` and the destination register `Sn` can contain either an integer or a single-precision floating-point number. Arithmetic instructions on the ARM treat the `Rd` value as an integer, whereas most VFP arithmetic instructions treat the `Fn` value as a single-precision floating-point number. If an integer is transferred, one of the integer-to-floating-point conversion instructions need to be executed after the `FMSR` instruction if subsequent VFP instructions are to yield sensible results.

Implementations are free to convert the value transferred to an internal format, provided they can recover either the correct single-precision value or the correct integer value (depending on how `Sn` is subsequently used).

C4.1.38 FMSRR

31	28	27 26 25 24 23 22 21 20	19	16	15	12	11 10 9 8	7	6	5	4	3	0
cond		1 1 0 0 0 1 0 0	Rn		Rd		1 0 1 1	0	R	M	1	Sm	
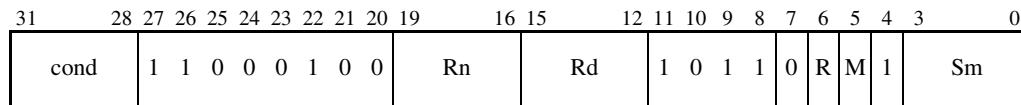
The FMSRR (Floating-point Move to two Single-precision Registers from two Registers) instruction transfers the contents of two ARM registers to a pair of single-precision VFP registers.

Syntax

```
FMSRR{<cond>}  {<Sm>, <Sn>}, <Rd, <Rn>
```

where:

<cond>	Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-7. If <cond> is omitted, the AL (always) condition is used.
<Sm>	Specifies the first destination single-precision VFP register. This is encoded in the instruction by setting Sm and M to the top 4 bits and the bottom bit respectively of m.
<Sn>	Specifies the second destination single-precision VFP register. Sn must be Sm+1.
<Rd>	Specifies the source ARM register for Sm.
<Rn>	Specifies the source ARM register for Sn.

Architecture version

D variants only, VFPv2 and above.

Exceptions

None

Operation

```
if ConditionPassed(cond) then
    Sm = Rd
    Sn = Rn
```

Notes

Conversions   In the programmer's model, FMSRR does not perform any conversion of the values transferred. Arithmetic instructions on either of the ARM registers treat the contents as an integer. Most VFP instructions treat the Sm and Sn values as single-precision floating-point numbers.

Invalid register lists

If S31 is specified as Sm, the results are UNPREDICTABLE. If the register pair is not consecutive, an error is reported by the assembler.

ARM DDI 0100F-02

C4.1.39 FMSTAT

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

The FMSTAT (Floating-point Move Status) instruction transfers the N, Z, C, and V flags in the FPSCR to the corresponding flags in the ARM's CPSR, and is normally used after one of the VFP comparison instructions has set the FPSCR flags.

Syntax

```
FMSTAT{<cond>}
```

where:

<cond>   Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-7. If <cond> is omitted, the AL (always) condition is used.

Architecture version

All

Exceptions

None

Operation

```
if ConditionPassed(cond) then
    CPSR N Flag = FPSCR N Flag
    CPSR Z Flag = FPSCR Z Flag
    CPSR C Flag = FPSCR C Flag
    CPSR V Flag = FPSCR V Flag
```

Notes

Encoding   The instruction FMSTAT{<cond>} is encoded as:

```
FMRX{<cond>}  r15, FPSCR
```

See also *FMRX* on page C4-63.

C4.1.40 FMULD

31	28	27	26	25	24	23	22	21	20	19	16	15	12	11	10	9	8	7	6	5	4	3	0
cond		1	1	1	0	0	0	1	0	Dn		Dd		1	0	1	1	0	0	0	0	Dm	
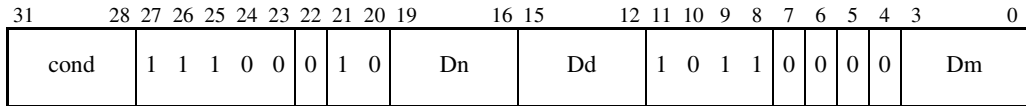
The FMULD (Floating-point Multiply, Double-precision) instruction multiplies together two double-precision registers and writes the result to a third double-precision register. It can also perform a vector version of this operation.

Syntax

```
FMULD{<cond>}  <Dd>, <Dn>, <Dm>
```

where:

<cond>	Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-7. If <cond> is omitted, the AL (always) condition is used.
<Dd>	Specifies the destination register.
<Dn>	Specifies the register that contains the first operand for the multiplication.
<Dm>	Specifies the register that contains the second operand for the multiplication.

Architecture version

D variants only

Exceptions

Floating-point exceptions: Invalid Operation, Overflow, Underflow, Inexact

Operation

```
if ConditionPassed(cond) then
    for i = 0 to vec_len-1
        Dd[i] = Dn[i] * Dm[i]
```

Notes

Vectors	When the LEN field of the FPSCR indicates scalar mode (vector length 1), FMULD performs one multiplication, and vec_len=1, Dd[0]=Dd, Dn[0]=Dn, and Dm[0]=Dm.
	When the LEN field indicates a vector mode (vector length > 1), FMULD might perform more than one multiplication. *Addressing Mode 2 - Double-precision vectors (non-monadic)* on page C5-8 describes how FMULD encodes the registers it uses and how vec_len, Dd[i], Dn[i], and Dm[i] are determined.

Rounding   This is a fully-rounded multiplication. The rounding mode is determined by the FPSCR.

C4.1.41 FMULS

31	28	27	26	25	24	23	22	21	20	19	16	15	12	11	10	9	8	7	6	5	4	3	0
cond		1	1	1	0	0	D	1	0	Fn		Fd		1	0	1	0	N	0	M	0	Fm	
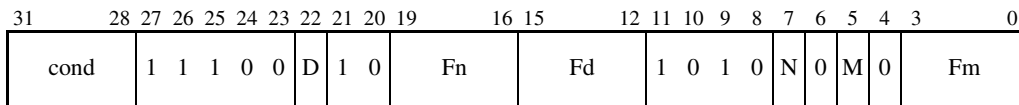
The `FMULS` (Floating-point Multiply, Single-precision) instruction multiplies together two single-precision registers and writes the result to a third single-precision register. It can also perform a vector version of this operation.

Syntax

```
FMULS{<cond>}  <Sd>, <Sn>, <Sm>
```

where:

<cond>   Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-7. If <cond> is omitted, the AL (always) condition is used.

<Sd>   Specifies the destination register. Its number is encoded as Fd (top 4 bits) and D (bottom bit).

<Sn>   Specifies the register that contains the first operand for the multiplication. Its number is encoded as Fn (top 4 bits) and N (bottom bit).

<Sm>   Specifies the register that contains the second operand for the multiplication. Its number is encoded as Fm (top 4 bits) and M (bottom bit).

Architecture version

All

Exceptions

Floating-point exceptions: Invalid Operation, Overflow, Underflow, Inexact

Operation

```
if ConditionPassed(cond) then
    for i = 0 to vec_len-1
        Sd[i] = Sn[i] * Sm[i]
```

Notes

Vectors When the LEN field of the FPSCR indicates scalar mode (vector length 1), FMULS performs just one multiplication, and `vec_len=1`, `Sd[0]=Sd`, Sn[0]=Sn, and `Sm[0]=Sm`.

When the LEN field indicates a vector mode (vector length > 1), FMULS might perform more than one multiplication. *Addressing Mode 1 - Single-precision vectors (non-monadic)* on page C5-2 shows how FMULS encodes the registers it uses and determines `vec_len`, `Sd[i]`, `Sn[i]`, and `Sm[i]`.

Rounding The operation is a fully-rounded multiplication. The rounding mode is determined by the FPSCR.

C4.1.42 FMXR

31	28	27	26	25	24	23	22	21	20	19		16	15		12	11	10	9	8	7	6	5	4	3		0
cond		1	1	1	0	1	1	1	0		reg			Rd		1	0	1	0	0	SBZ		1		SBZ	

The FMXR (Floating-point Move to System Register from Register) instruction transfers the contents of the ARM register Rd to one of the VFP system registers.

Syntax

```
FMXR{<cond>}  <reg>, <Rd>
```

where:

<cond>   Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-7. If <cond> is omitted, the AL (always) condition is used.

<reg>   Specifies the destination system register as follows:

```
<reg> = 0b0000: FPSID
<reg> = 0b0001: FPSCR
<reg> = 0b1000: FPEXC
```

Other values of <reg> can be used by individual VFP implementations for IMPLEMENTATION DEFINED purposes. Typically, they are used to transfer data to a hardware coprocessor from the support code for that coprocessor.

All other code must treat such values of <reg> as UNPREDICTABLE and not to be relied upon.

<Rd>   Specifies the source ARM register.

Architecture version

All

Exceptions

None

Operation

```
if ConditionPassed(cond) then
    reg = Rd
```

Notes

Serialization   FMXR is a serializing instruction. See *System register transfer instructions* on page C3-20 for details of what this means.

Exception processing

After serialization, if the VFP system contains a hardware coprocessor, that coprocessor might have a pending exception to process. Whether the FMXR instruction triggers the processing of such an instruction depends on which system register is being transferred, as described in the following notes. If exception processing is triggered, this causes the FMXR instruction to take the ARM's Undefined Instruction exception.

Writing FPSID

An FMXR instruction with destination FPSID can be executed in any ARM processor mode. It is a serializing no-op, because FPSID is a read-only register, and does not trigger exception processing.

Writing FPSCR

An FMXR instruction with destination FPSCR can be executed in any ARM processor mode. After serialization, exception processing is triggered if necessary. Otherwise, the value of Rd is written to the FPSCR.

Writing FPEXC

An FMXR instruction with destination FPEXC can only be executed in privileged ARM processor modes. An attempt to execute it in User mode causes the ARM's Undefined Instruction exception to be taken.

After serialization, it writes the value of Rd to FPEXC, and does not trigger exception processing. Because all but bits[31:30] of FPEXC is IMPLEMENTATION DEFINED, non implementation-specific code must only use such an instruction as part of a read/modify bits[31:0]/write sequence.

C4.1.43 FNEGD

31				28	27	26	25	24	23	22	21	20	19			16	15			12	11	10	9	8	7	6	5	4	3			0
cond					1	1	1	0	1	0	1	1	0	0	0	1		Dd			1	0	1	1	0	1	0	0		Dm		
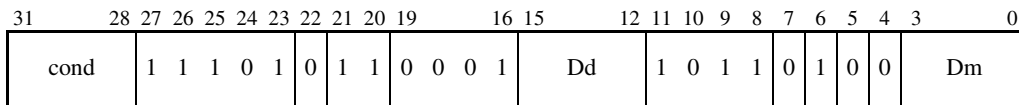
The FNEGD (Floating-point Negate, Double-precision) instruction negates the value of a double-precision register and writes the result to another double-precision register. It can also perform a vector version of this operation.

Syntax

```
FNEGD{<cond>}  <Dd>, <Dm>
```

where:

<cond>   Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-7. If <cond> is omitted, the AL (always) condition is used.

<Dd>   Specifies the destination register.

<Dm>   Specifies the source register.

Architecture version

D variants only

Exceptions

None

Operation

```
if ConditionPassed(cond) then
    for i = 0 to vec_len-1
        Dd[i] = -(Dm[i])
```

Notes

Negation The function - (x) means a copy of x with its sign bit reversed, as defined in the Appendix
to the IEEE 754-1985 standard.

Flush-to-zero mode

  The FZ bit of the FPSCR does not affect the operand or result of this instruction.

Vectors When the LEN field of the FPSCR indicates scalar mode (vector length 1), FNEGD performs
just one negation operation, and vec_len=1, Dd[0]=Dd, and Dm[0]=Dm.

  When the LEN field indicates a vector mode (vector length > 1), FNEGD might perform
more than one negation operation. *Addressing Mode 4 - Double-precision vectors (monadic)*
on page C5-19 shows how FNEGD encodes its registers and determines the values of
vec_len, Dd[i], and Dm[i].

Signaling NaNs

  To comply with the VFP architecture, FNEGD must not generate an exception even if the
value in its source register is a signaling NaN. This is a more stringent requirement than the
one in the Appendix to the IEEE 754-1985 standard.

C4.1.44 FNEGS

31		28	27	26	25	24	23	22	21	20	19			16	15			12	11	10	9	8	7	6	5	4	3		0
cond			1	1	1	0	1	D	1	1	0	0	0	1		Fd			1	0	1	0	0	1	M	0		Fm	
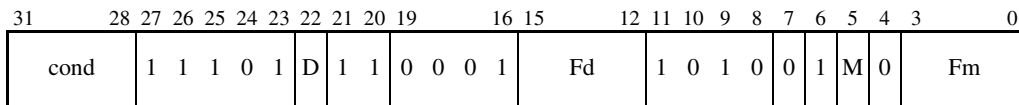
The `FNEGS` (Floating-point Negate, Single-precision) instruction negates the value of a single-precision register and writes the result to another single-precision register. It can also perform a vector version of this operation.

Syntax

```
FNEGS{<cond>}  <Sd>, <Sm>
```

where:

`<cond>`	Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-7. If `<cond>` is omitted, the AL (always) condition is used.
`<Sd>`	Specifies the destination register. Its number is encoded as Fd (top 4 bits) and D (bottom bit).
`<Sm>`	Specifies the source register. Its number is encoded as Fm (top 4 bits) and M (bottom bit).

Architecture version

All

Exceptions

None

Operation

```
if ConditionPassed(cond) then
    for i = 0 to vec_len-1
        Sd[i] = -(Sm[i])
```

Notes

Negation   The function - (x) means a copy of x with its sign bit reversed, as defined in the Appendix to the IEEE 754-1985 standard.

Flush-to-zero mode

The FZ bit of the FPSCR does not affect the operand or result of this instruction.

Vectors   When the LEN field of the FPSCR indicates scalar mode (vector length 1), FNEGS performs just one negation operation, and vec_len=1, Sd[0]=Sd, and Sm[0]=Sm.

When the LEN field indicates a vector mode (vector length > 1), FNEGS might perform more than one negation operation. *Addressing Mode 3 - Single-precision vectors (monadic)* on page C5-14 shows how FNEGS encodes its registers and determines vec_len, Sd[i], and Sm[i].

Signaling NaNs

To comply with the VFP architecture, FNEGS must not generate an exception even if the value in its source register is a signaling NaN. This is a more stringent requirement than the one in the Appendix to the IEEE 754-1985 standard.

C4.1.45 FNMACD

31		28	27	26	25	24	23	22	21	20	19		16	15		12	11	10	9	8	7	6	5	4	3		0
cond			1	1	1	0	0	0	0	0		Dn			Dd		1	0	1	1	0	1	0	0		Dm	
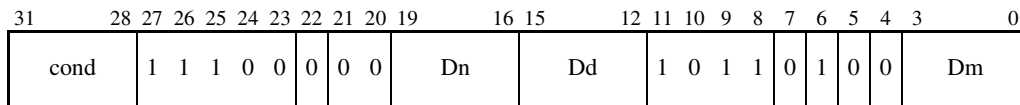
The FNMACD (Floating-point Negated Multiply and Accumulate, Double-precision) instruction multiplies together two double-precision registers, adds a third double-precision register to the negation of the product and writes the result to the third register. It can also perform a vector version of this operation.

Syntax

FNMACD{<cond>}   <Dd>, <Dn>, <Dm>

where:

<cond>	Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-7. If <cond> is omitted, the AL (always) condition is used.
<Dd>	Specifies the destination register, which is also used as the second operand for the addition.
<Dn>	Specifies the register that contains the first operand for the multiplication.
<Dm>	Specifies the register that contains the second operand for the multiplication.

Architecture version

D variants only

Exceptions

Floating-point exceptions: Invalid Operation, Overflow, Underflow, Inexact

Operation

```
if ConditionPassed(cond) then
    for i = 0 to vec_len-1
        Dd[i] = -(Dn[i] * Dm[i]) + Dd[i]
```

Notes

Vectors   When the LEN field of the FPSCR indicates scalar mode (vector length 1), FNMACD
performs just one multiply-negate-add operation, and `vec_len=1`, `Dd[0]=Dd`,
`Dn[0]=Dn`, and `Dm[0]=Dm`.

When the LEN field indicates a vector mode (vector length > 1), FNMACD might perform
more than one multiply-negate-add operation. *Addressing Mode 4 - Double-precision
vectors (monadic)* on page C5-19 shows how FNMACD encodes its registers and determines
`vec_len`, `Dd[i]`, `Dn[i]`, and `Dm[i]`.

Rounding   The operation is a fully-rounded multiplication with the rounding mode determined by the
FPSCR, followed by reversal of the sign bit and a fully-rounded addition, using the same
rounding mode.

C4.1.46 FNMACS

31	28	27 26 25 24 23	22	21 20	19	16	15	12	11 10 9 8	7	6	5	4	3	0
cond		1 1 1 0 0	D	0 0	Fn		Fd		1 0 1 0	N	1	M	0	Fm	

The FNMACS (Floating-point Negated Multiply and Accumulate, Single-precision) instruction multiplies together two single-precision registers, adds a third single-precision register to the negation of the product and writes the result to the third register. It can also perform a vector version of this.

Syntax

```
FNMACS{<cond>}  <Sd>, <Sn>, <Sm>
```

where:

<cond>	Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-7. If <cond> is omitted, the AL (always) condition is used.
<Sd>	Specifies the destination register, which is also used as the second operand for the addition. Its number is encoded as Fd (top 4 bits) and D (bottom bit).
<Sn>	Specifies the register that contains the first operand for the multiplication. Its number is encoded as Fn (top 4 bits) and N (bottom bit).
<Sm>	Specifies the register that contains the second operand for the multiplication. Its number is encoded as Fm (top 4 bits) and M (bottom bit).

Architecture version

All

Exceptions

Floating-point exceptions: Invalid Operation, Overflow, Underflow, Inexact
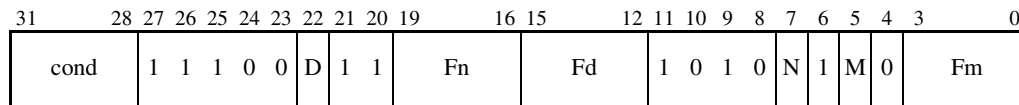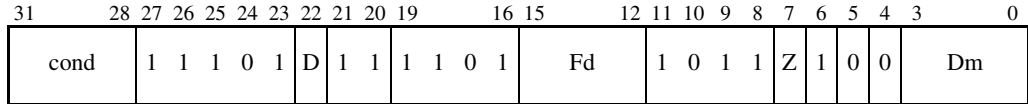
Operation

```
if ConditionPassed(cond) then
    for i = 0 to vec_len-1
        Sd[i] = -(Sn[i] * Sm[i]) + Sd[i]
```

Notes

Vectors   When the LEN field of the FPSCR indicates scalar mode (vector length 1), FNMACS
performs just one multiply-negate-add operation, and `vec_len=1`, `Sd[0]=Sd`,
`Sn[0]=Sn`, and `Sm[0]=Sm`.

When the LEN field indicates a vector mode (vector length > 1), FNMACS might perform
more than one multiply-negate-add operation. *Addressing Mode 1 - Single-precision vectors
(non-monadic)* on page C5-2 describes how FNMACS encodes the registers it uses and how
`vec_len`, `Sd[i]`, `Sn[i]`, and `Sm[i]` are determined.

Rounding   The operation is a fully-rounded multiplication with the rounding mode determined by the
FPSCR, followed by reversal of the sign bit and a fully-rounded addition, using the same
rounding mode.

C4.1.47 FNMSCD

31	28	27	26	25	24	23	22	21	20	19	16	15	12	11	10	9	8	7	6	5	4	3	0
cond		1	1	1	0	0	0	0	1	Dn		Dd		1	0	1	1	0	1	0	0	Dm	

The FNMSCD (Floating-point Negated Multiply and Subtract, Double-precision) instruction multiplies together two double-precision registers, subtracts a third double-precision register from the negation of the product and writes the result to the third register. It can also perform a vector version of this operation.

Syntax

```
FNMSCD{<cond>}  <Dd>, <Dn>, <Dm>
```

where:

<cond>   Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-7. If <cond> is omitted, the AL (always) condition is used.

<Dd>   Specifies the destination register, and is also used as the second operand for the subtraction.

<Dn>   Specifies the register that contains the first operand for the multiplication.

<Dm>   Specifies the register that contains the second operand for the multiplication.

Architecture version

D variants only

Exceptions

Floating-point exceptions: Invalid Operation, Overflow, Underflow, Inexact

Operation

```
if ConditionPassed(cond) then
    for i = 0 to vec_len-1
        Dd[i] = -(Dn[i] * Dm[i]) - Dd[i]
```

Notes

Vectors   When the LEN field of the FPSCR indicates scalar mode (vector length 1), `FNMSCD` performs just one multiply-negate-subtract operation, and `vec_len=1`, `Dd[0]=Dd`, `Dn[0]=Dn`, and `Dm[0]=Dm`.

When the LEN field indicates a vector mode (vector length > 1), `FNMSCD` might perform more than one multiply-negate-subtract operation. *Addressing Mode 2 - Double-precision vectors (non-monadic)* on page C5-8 describes how `FNMSCD` encodes the registers it uses and how `vec_len`, `Dd[i]`, `Dn[i]`, and `Dm[i]` are determined.

Rounding   The operation is a fully-rounded multiplication with the rounding mode determined by the FPSCR, followed by reversal of the sign bit and a fully-rounded subtraction, using the same rounding mode.

C4.1.48 FNMSCS

31		28	27	26	25	24	23	22	21	20	19		16	15		12	11	10	9	8	7	6	5	4	3		0
cond			1	1	1	0	0	D	0	1	Fn			Fd			1	0	1	0	N	1	M	0	Fm		
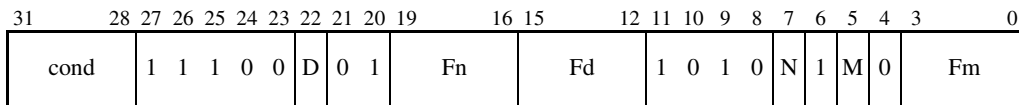
The FNMSCS Floating-point Negated Multiply and Subtract, Single-precision() instruction multiplies together two single-precision registers, subtracts a third single-precision register from the negation of the product and writes the result to the third register. It can also perform a vector version of this operation.

Syntax

```
FNMSCS{<cond>}  <Sd>, <Sn>, <Sm>
```

where:

`<cond>`	Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-7. If `<cond>` is omitted, the AL (always) condition is used.
`<Sd>`	Specifies the destination register, which is also used as the second operand for the subtraction. The register number is encoded as Fd (top 4 bits) and D (bottom bit).
`<Sn>`	Specifies the register that contains the first operand for the multiplication. The register number is encoded as Fn (top 4 bits) and N (bottom bit).
`<Sm>`	Specifies the register that contains the second operand for the multiplication. The register number is encoded as Fm (top 4 bits) and M (bottom bit).

Architecture version

All

Exceptions

Floating-point exceptions: Invalid Operation, Overflow, Underflow, Inexact

Operation

```
if ConditionPassed(cond) then
    for i = 0 to vec_len-1
        Sd[i] = -(Sn[i] * Sm[i]) - Sd[i]
```

Notes

Vectors   When the LEN field of the FPSCR indicates scalar mode (vector length 1), `FNMSCS`
performs just one multiply-negate-subtract operation, and `vec_len=1`, `Sd[0]=Sd`,
`Sn[0]=Sn`, and `Sm[0]=Sm`.

When the LEN field indicates a vector mode (vector length > 1), `FNMSCS` might perform
more than one multiply-negate-subtract operation. *Addressing Mode 1 - Single-precision
vectors (non-monadic)* on page C5-2 describes how `FNMSCS` encodes the registers it uses
and how `vec_len`, `Sd[i]`, `Sn[i]`, and `Sm[i]` are determined.

Rounding   The operation is a fully-rounded multiplication with the rounding mode determined by the
FPSCR, followed by reversal of the sign bit and a fully-rounded subtraction, using the same
rounding mode.

C4.1.49 FNMULD

31			28	27	26	25	24	23	22	21	20	19			16	15			12	11	10	9	8	7	6	5	4	3			0
cond				1	1	1	0	0	0	1	0	Dn				Dd				1	0	1	1	0	1	0	0	Dm			
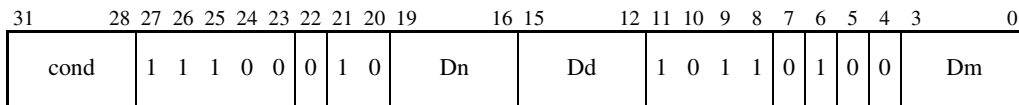
The FNMULD (Floating-point Negated Multiply, Double-precision) instruction multiplies together two double-precision registers, and writes the negation of the result to a third double-precision register. It can also perform a vector version of this operation.

Syntax

```
FNMULD{<cond>}  <Dd>, <Dn>, <Dm>
```

where:

<cond>   Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-7. If <cond> is omitted, the AL (always) condition is used.

<Dd>   Specifies the destination register.

<Dn>   Specifies the register that contains the first operand for the multiplication.

<Dm>   Specifies the register that contains the second operand for the multiplication.

Architecture version

D variants only

Exceptions

Floating-point exceptions: Invalid Operation, Overflow, Underflow, Inexact

Operation

```
if ConditionPassed(cond) then
    for i = 0 to vec_len-1
        Dd[i] = -(Dn[i] * Dm[i])
```

Notes

Vectors   When the LEN field of the FPSCR indicates scalar mode (vector length 1), FNMULD
performs just one negated multiplication, and vec_len=1, Dd[0]=Dd, Dn[0]=Dn, and
Dm[0]=Dm.

When the LEN field indicates a vector mode (vector length > 1), FNMULD might perform
more than one negated multiplication. *Addressing Mode 2 - Double-precision vectors
(non-monadic)* on page C5-8 describes how FNMULD encodes the registers it uses and how
vec_len, Dd[i], Dn[i], and Dm[i] are determined.

Rounding   The operation is a fully-rounded multiplication. The rounding mode is determined by the
FPSCR, followed by reversal of the sign bit of the result.

C4.1.50 FNMULS

31		28	27	26	25	24	23	22	21	20	19		16	15		12	11	10	9	8	7	6	5	4	3		0
	cond		1	1	1	0	0	D	1	0		Fn			Fd		1	0	1	0	N	1	M	0		Fm	
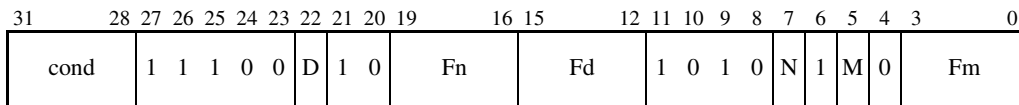
The FNMULS (Floating-point Negated Multiply, Single-precision) instruction multiplies together two single-precision registers, and writes the negation of the result to a third single-precision register. It can also perform a vector version of this operation.

Syntax

FNMULS{<cond>} <Sd>, <Sn>, <Sm>

where:

<cond>	Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-7. If <cond> is omitted, the AL (always) condition is used.
<Sd>	Specifies the destination register. The register number is encoded as Fd (top 4 bits) and D (bottom bit).
<Sn>	Specifies the register that contains the first operand for the multiplication. The register number is encoded as Fn (top 4 bits) and N (bottom bit).
<Sm>	Specifies the register that contains the second operand for the multiplication. The register number is encoded as Fm (top 4 bits) and M (bottom bit).

Architecture version

All

Exceptions

Floating-point exceptions: Invalid Operation, Overflow, Underflow, Inexact

Operation

```
if ConditionPassed(cond) then
    for i = 0 to vec_len-1
        Sd[i] = -(Sn[i] * Sm[i])
```

Notes

Vectors   When the LEN field of the FPSCR indicates scalar mode (vector length 1), FNMULS
performs just one negated multiplication, and `vec_len=1`, `Sd[0]=Sd`, `Sn[0]=Sn`, and
`Sm[0]=Sm`.

When the LEN field indicates a vector mode (vector length > 1), FNMULS might perform
more than one negated multiplication. *Addressing Mode 1 - Single-precision vectors
(non-monadic)* on page C5-2 shows how FNMULS encodes its registers and determines
`vec_len`, `Sd[i]`, `Sn[i]`, and `Sm[i]`.

Rounding   The operation is a fully-rounded multiplication. The rounding mode is determined by the
FPSCR, followed by reversal of the sign bit of the result.

C4.1.51 FSITOD

31			28	27	26	25	24	23	22	21	20	19			16	15			12	11	10	9	8	7	6	5	4	3			0
cond				1	1	1	0	1	0	1	1	1	0	0	0	Dd				1	0	1	1	1	1	M	0	Fm			

The FSITOD (Floating-point Convert Signed Integer to Double-precision) instruction converts a signed integer value held in a single-precision register to double-precision and writes the result to a double-precision register. The integer value will normally have been transferred from memory by a single-precision load instruction or from an ARM register by an FMSR instruction.

Syntax

```
FSITOD{<cond>}  <Dd>, <Sm>
```

where:

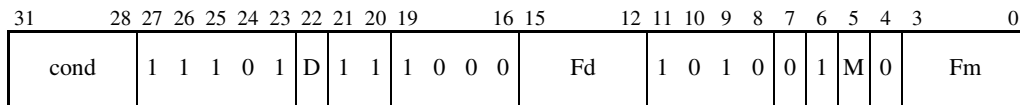|	
<cond>	Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-7. If <cond> is omitted, the AL (always) condition is used.
<Dd>	Specifies the destination register.
<Sm>	Specifies the source register. The register number is encoded as Fm (top 4 bits) and M (bottom bit).

Architecture version

D variants only

Exceptions

None

Operation

```
if ConditionPassed(cond) then
    Dd = ConvertSignedIntegerToDouble(Sm)
```

Notes

Vectors	FSITOD always specifies a scalar operation, regardless of the LEN field of the FPSCR.
Zero	If Sm contains an integer zero, the result is a double-precision +0.0, not a double-precision -0.0.

C4.1.52 FSITOS

31			28	27	26	25	24	23	22	21	20	19			16	15			12	11	10	9	8	7	6	5	4	3			0
cond				1	1	1	0	1	D	1	1	1	0	0	0	Fd				1	0	1	0	1	1	M	0	Fm			
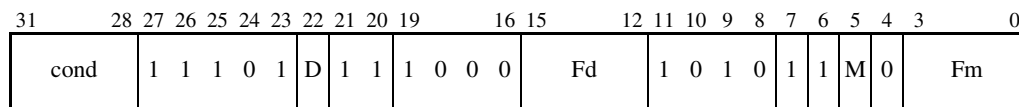
The FSITOS (Floating-point Convert Signed Integer to Single-precision) instruction converts a signed
integer value held in a single-precision register to single-precision and writes the result to a second
single-precision register. The integer value will normally have been transferred from memory by a
single-precision load instruction or from an ARM register by an FMSR instruction.

Syntax

```
FSITOS{<cond>}  <Sd>, <Sm>
```

where:

<cond>	Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-7. If <cond> is omitted, the AL (always) condition is used.
<Sd>	Specifies the destination register. The register number is encoded as Fd (top 4 bits) and D (bottom bit).
<Sm>	Specifies the source register. The register number is encoded as Fm (top 4 bits) and M (bottom bit).

Architecture version

All

Exceptions

Floating-point exception: Inexact

Operation

```
if ConditionPassed(cond) then
    Sd = ConvertSignedIntegerToSingle(Sm)
```

Notes

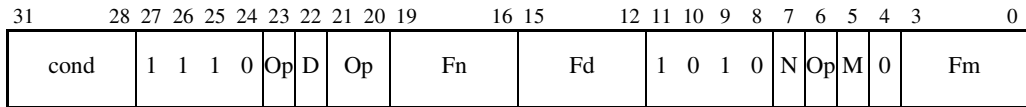|	
Vectors	FSITOS always specifies a scalar operation, regardless of the LEN field of the FPSCR.
Zero	If Sm contains an integer zero, the result is a single-precision +0.0, not a single-precision -0.0.
Rounding	Rounding is needed for some large operand values. The rounding mode is determined by the FPSCR.

C4.1.53 FSQRTD

31			28	27	26	25	24	23	22	21	20	19			16	15			12	11	10	9	8	7	6	5	4	3		0
cond				1	1	1	0	1	0	1	1	0	0	0	1	Dd				1	0	1	1	1	1	0	0	Dm		

The FSQRTD (Floating-point Square Root, Double-precision) instruction calculates the square root of the value in a double-precision register and writes the result to another double-precision register. It can also perform a vector version of this operation.

Syntax

```
FSQRTD{<cond>}  <Dd>, <Dm>
```

where:

<cond>   Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-7. If <cond> is omitted, the AL (always) condition is used.

<Dd>   Specifies the destination register.

<Dm>   Specifies the source register.

Architecture version

D variants only

Exceptions

Floating-point exceptions: Invalid Operation, Inexact

Operation

```
if ConditionPassed(cond) then
    for i = 0 to vec_len-1
        Dd[i] = sqrt(Dm[i])
```

Usage

Square roots take a large number of cycles on most implementations, and vector square roots take proportionately longer. This can have a major effect on performance, and so the use of large numbers of square roots should be avoided where possible.

Also see *Interrupts* on page C1-6 for a description of some implications for interrupt latency.

Notes

Vectors   When the LEN field of the FPSCR indicates scalar mode (vector length 1), FSQRTD
performs just one square root operation, and vec_len=1, Dd[0]=Dd, and Dm[0]=Dm.

When the LEN field indicates a vector mode (vector length > 1), FSQRTD might perform
more than one square root operation. *Addressing Mode 4 - Double-precision vectors
(monadic)* on page C5-19 describes how FSQRTD encodes the registers it uses and how
vec_len, Dd[i], and Dm[i] are determined.

Rounding   The operation is a fully-rounded square root operation. The rounding mode is determined
by the FPSCR.

C4.1.54 FSQRTS

31	28	27	26	25	24	23	22	21	20	19			16	15			12	11	10	9	8	7	6	5	4	3			0
cond		1	1	1	0	1	D	1	1	0	0	0	1		Fd			1	0	1	0	1	1	M	0		Fm	

The `FSQRTS` (Floating-point Square Root, Single-precision) instruction calculates the square root of the value in a single-precision register and writes the result to another single-precision register. It can also perform a vector version of this operation.

Syntax

```
FSQRTS{<cond>}  <Sd>, <Sm>
```

where:

<cond>   Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-7. If `<cond>` is omitted, the AL (always) condition is used.

<Sd>   Specifies the destination register. The register number is encoded as Fd (top 4 bits) and D (bottom bit).

<Sm>   Specifies the source register. The register number is encoded as Fm (top 4 bits) and M (bottom bit).

Architecture version

All

Exceptions

Floating-point exceptions: Invalid Operation, Inexact

Operation

```
if ConditionPassed(cond) then
    for i = 0 to vec_len-1
        Sd[i] = sqrt(Sm[i])
```

Usage

Square roots take a large number of cycles on most implementations, and vector square roots take proportionately longer. This can have a major effect on performance, and so the use of large numbers of square roots should be avoided where possible.

Also see *Interrupts* on page C1-6 for a description of some implications for interrupt latency.

Notes

Vectors When the LEN field of the FPSCR indicates scalar mode (vector length 1), FSQRTS performs just one square root operation, and vec_len=1, Sd[0]=Sd, and Sm[0]=Sm.

When the LEN field indicates a vector mode (vector length > 1), FSQRTS might perform more than one square root operation. *Addressing Mode 3 - Single-precision vectors (monadic)* on page C5-14 describes how FSQRTS encodes the registers it uses and how vec_len, Sd[i], and Sm[i] are determined.

Rounding This is a fully-rounded square root operation. The FPSCR determines the rounding mode.

C4.1.55 FSTD

31		28 27	26	25	24	23	22	21	20	19		16	15		12	11	10	9	8	7	6	5	4	3		0
cond		1	1	0	1	U	0	0	0		Rn			Dd		1	0	1	1				offset			
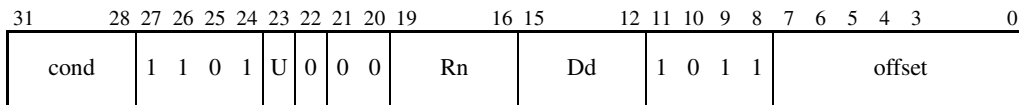
The FSTD (Floating-point Store, Double-precision) instruction stores a double-precision register to memory.

Syntax

```
FSTD{<cond>}  <Dd>, [<Rn>{, #+/-(<offset>*4)}]
```

where:

<cond>   Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-7. If <cond> is omitted, the AL (always) condition is used.

<Dd>   Specifies the source register.

<Rn>   Specifies the register holding the base address for the transfer.

<offset>   Specifies an offset to be multiplied by 4, then added to the base address (if U == 1) or subtracted from it (if U == 0) to form the actual address of the transfer. If offset is omitted, it defaults to +0.

Architecture version

D variants only

Exceptions

Data Abort

Operation

```
if ConditionPassed(cond) then
    if (U == 1)
        address = Rn + offset * 4
    else
        address = Rn - offset * 4
    if (big-endian)
        Memory[address,4] = Dd[63:32]
        Memory[address+4,4] = Dd[31:0]
    else
        Memory[address,4] = Dd[31:0]
        Memory[address+4,4] = Dd[63:32]
```

Notes

Addressing mode

This is a special case of *Addressing Mode 5 - VFP load/store multiple* on page C5-24.

Conversions An implementation using an internal format for double-precision values must convert that format back to the external double-precision format. Otherwise, no conversion is required.

C4.1.56 FSTMD

31		28	27	26	25	24	23	22	21	20	19			16	15			12	11	10	9	8	7	6	5	4	3			0
cond			1	1	0	P	U	0	W	0		Rn				Dd			1	0	1	1			offset					
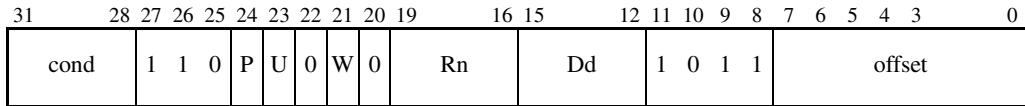
The FSTMD (Floating-point Store Multiple, Double-precision) instruction stores a sequence of consecutive double-precision registers to memory.

Syntax

```
FSTM<addressing_mode>D{<cond>}  <Rn>{!}, <registers>
```

where:

<cond>   Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-7. If <cond> is omitted, the AL (always) condition is used.

<addressing_mode>

Specifies the addressing mode, which determines the values of start_address and end_address used by the instruction. See *Addressing Mode 5 - VFP load/store multiple* on page C5-24 for details.

<Rn>   Specifies the base register used by <addressing_mode>.

!   Sets the W bit of the instruction to 1, specifying that the base register <Rn> is to be updated by the instruction. If it is omitted, the W bit of the instruction is set to 0 and the base register <Rn> is left unchanged. Some combinations of <addressing_mode> and the presence or absence of ! are not allowed. For details, see *Addressing Mode 5 - VFP load/store multiple* on page C5-24.

<registers>

Specifies which registers are to be stored, as a list of consecutively numbered double-precision registers, separated by commas and surrounded by brackets. It is encoded in the instruction by setting Dd to the number of the first register in the list, and offset to twice the number of registers in the list. At least one register must be specified in the list.

For example, if <registers> is {D2,D3,D4}, the Dd field of the instruction is 2 and the offset field is 6.

Architecture version

D variants only

Exceptions

Data Abort

Operation

```
if ConditionPassed(cond) then
    address = start_address
    for i = 0 to (offset-2)/2
        /* d is the number of register Dd;                */
        /* D(n) is the double-precision register numbered n */
        if (big-endian)
            Memory[address,4] = D(d+i)[63:32]
            Memory[address+4,4] = D(d+i)[31:0]
        else
            Memory[address,4] = D(d+i)[31:0]
            Memory[address+4,4] = D(d+i)[63:32]
        address = address + 8
    assert end_address = address - 4
```

Notes

Encoding If P=1 and W=0, the instruction is instead an FSTD instruction. Otherwise, if offset is odd, the instruction is instead an FSTMX instruction.

Vectors The FSTMD instruction is unaffected by the LEN and STRIDE fields of the FPSCR, and does not wrap around at bank boundaries in the way that vector operands to data-processing instructions do. Registers are stored in simple increasing order of register number.

Invalid register lists

If Dd and offset do not specify a valid register list, the instruction is UNPREDICTABLE. This happens in two cases:

- if offset == 0, that is, if an attempt is made to transfer no registers

- if d + offset/2 > 16, that is, if an attempt is made to transfer another register after D15.

Conversions If an implementation uses an internal format for double-precision values, it must convert that format back to the external double-precision format. Otherwise, no conversion is required.

C4.1.57 FSTMS

31	28	27	26	25	24	23	22	21	20	19	16	15	12	11	10	9	8	7		0
cond		1	1	0	P	U	D	W	0	Rn		Fd		1	0	1	0		offset	
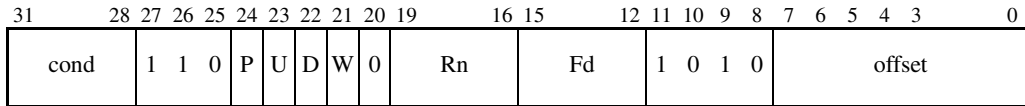
The FSTMS (Floating-point Store Multiple, Single-precision) instruction stores a sequence of consecutive single-precision registers to memory.

Syntax

```
FSTM<addressing_mode>S{<cond>}  <Rn>{!}, <registers>
```

where:

<cond> Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-7. If <cond> is omitted, the AL (always) condition is used.

<addressing_mode>

Specifies the addressing mode, which determines the values of start_address and end_address used by the instruction. See *Addressing Mode 5 - VFP load/store multiple* on page C5-24.

<Rn> Specifies the base register used by <addressing_mode>.

! Sets the W bit of the instruction to 1, specifying that the base register <Rn> is to be updated by the instruction. If it is omitted, the W bit of the instruction is set to 0 and the base register <Rn> is left unchanged. Some combinations of <addressing_mode> and the presence or absence of ! are not allowed. For details, see *Addressing Mode 5 - VFP load/store multiple* on page C5-24.

<registers>

Specifies which registers are to be stored, as a list of consecutively numbered single-precision registers, separated by commas and surrounded by brackets. If d is the number of the first register in the list, the list is encoded in the instruction by setting Fd and D to the top 4 bits and the bottom bit respectively of d, and offset to the number of registers in the list. At least one register must be specified in the list.

For example, if <registers> is {S5,S6,S7}, the Fd field of the instruction is 0b0010, the D bit is 1 and the offset field is 3.

Architecture version

All

Exceptions

Data Abort

Operation

```
if ConditionPassed(cond) then
    address = start_address
    for i = 0 to offset-1
        /* d is as defined for <registers> above;        */
        /* S(n) is the single-precision register numbered n */
        Memory[address,4] = S(d+i)
        address = address + 4
    assert end_address = address - 4
```

Notes

Encoding   If P=1 and W=0, the instruction is instead an FSTS instruction.

Vectors   The FSTMS instruction is unaffected by the LEN and STRIDE fields of the FPSCR, and does not wrap around at bank boundaries in the way that vector operands to data-processing instructions do. Registers are stored in simple increasing order of register number.

Invalid register lists

If Fd, Dd and offset do not specify a valid register list, the instruction is UNPREDICTABLE. This happens in two cases:

- if offset == 0, that is, if an attempt is made to transfer no registers

- if d + offset > 32, that is, if an attempt is made to transfer another register after S31.

Conversions   In the programmer's model is that FSTMS does not perform any conversion on the value transferred. The source registers can each contain either a single-precision floating-point number or an integer. The latter is typically obtained as the result of one of the floating-point-to-integer conversion instructions.

Implementations are free to hold the values in the source registers in an internal format, provided that FSTMS converts it to external format and this conversion recovers the correct data, regardless of whether the register contains a single-precision floating-point number or an integer.

C4.1.58 FSTMX

31	28	27	26	25	24	23	22	21	20	19	16	15	12	11	10	9	8	7					0
cond		1	1	0	P	U	0	W	0	Rn		Dd		1	0	1	1			offset			
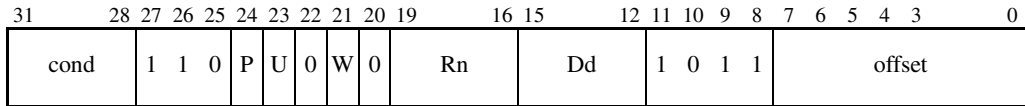
The FSTMX (Floating-point Store Multiple, Unknown precision) instruction stores a sequence of consecutive double-precision registers to memory, in an IMPLEMENTATION DEFINED format that allows the registers to be reloaded correctly regardless of whether they contain integers, single-precision values or double-precision values.

Syntax

```
FSTM<addressing_mode>X{<cond>}  <Rn>{!}, <registers>
```

where:

<cond>   Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-7. If <cond> is omitted, the AL (always) condition is used.

<addressing_mode>

                Specifies the addressing mode, which determines the values of start_address and end_address used by the instruction. See *Addressing Mode 5 - VFP load/store multiple* on page C5-24 for details.

<Rn>   Specifies the base register used by <addressing_mode>.

!   Sets the W bit of the instruction to 1, specifying that the base register <Rn> is to be updated by the instruction. If it is omitted, the W bit of the instruction is set to 0 and the base register <Rn> is left unchanged. Some combinations of <addressing_mode> and the presence or absence of ! are not allowed. For details, see *Addressing Mode 5 - VFP load/store multiple* on page C5-24.

<registers>

                Specifies which registers are to be stored, as a list of consecutively numbered double-precision registers, separated by commas and surrounded by brackets. It is encoded in the instruction by setting Dd to the number of the first register in the list, and offset to twice the number of registers in the list plus 1. At least one register must be named in the list.

                For example, if <registers> is {D2,D3,D4}, the Dd field of the instruction is 2 and the offset field is 7.

Architecture version

All

Exceptions

Data Abort

Operation

```
if ConditionPassed(cond) then
    /* d is the number of register Dd;                   */
    /* D(n) is the double-precision register numbered n */
    Store registers D(d) to D(d+(offset-3)/2) to memory words
      Memory[start_address,4] through to Memory[end_address-4,4]
```

Usage

FSTMX is used to save VFP register values to memory in circumstances where it is unknown what type of data they contain. Typical cases of this are:

•   in procedure entry sequences when a callee-save procedure calling standard is being used

•   in process swap code.

Notes

Encoding   If P=1 and W=0, the instruction is instead an FSTD instruction. Otherwise, if offset is even, the instruction is instead an FSTMD instruction.

Vectors   The FSTMX instruction is unaffected by the LEN and STRIDE fields of the FPSCR, and does not wrap around at bank boundaries in the way that vector operands to data-processing instructions do. Registers are stored in simple increasing order of register number.

Invalid register lists

If Dd and offset do not specify a valid register list, the instruction is UNPREDICTABLE. This happens in two cases:

•   if offset == 0, that is, if an attempt is made to transfer no registers

•   if d + (offset-1)/2 > 16, that is, if an attempt is made to transfer another register after D15.

Memory format

The memory format used is IMPLEMENTATION DEFINED. The only requirements on it are:

•   A matching FLDMX instruction must reload the registers correctly, regardless of whether they originally contained integers, single-precision values or double-precision values. For an FLDMX instruction and an FSTMX instruction to count as matching, they must specify identical register lists and must generate the same start_address and end_address when executed.

•   It must not use more than the allowed number of memory words, that is, 2N+1 memory words for storing N double-precision registers.

Non implementation-specific code must not do anything with the results of an FSTMX except reload it with a matching FLDMX instruction.

C4.1.59 FSTS

31		28	27	26	25	24	23	22	21	20	19		16	15		12	11	10	9	8	7	6	5	4	3		0
cond			1	1	0	1	U	D	0	0		Rn			Fd		1	0	1	0			offset				
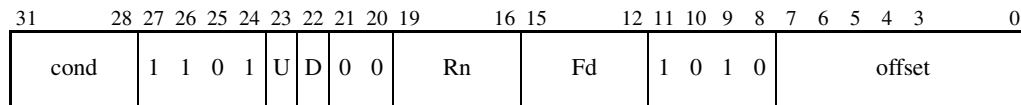
The FSTS (Floating-point Store, Single-precision) instruction stores a single-precision register to memory.

Syntax

```
FSTS{<cond>}  <Sd>, [<Rn>{, #+/-(<offset>*4)}]
```

where:

<cond>   Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-7. If <cond> is omitted, the AL (always) condition is used.

<Sd>   Specifies the source register. Its number is encoded as Fd (top 4 bits) and D (bottom bit).

<Rn>   Specifies the register holding the base address for the transfer.

<offset>   Specifies an offset to be multiplied by 4, then added to the base address (if U == 1) or subtracted from it (if U == 0) to form the actual address of the transfer. If this offset is omitted, it defaults to +0.

Architecture version

All

Exceptions

Data Abort

Operation

```
if ConditionPassed(cond) then
    if (U == 1)
        address = Rn + offset * 4
    else
        address = Rn - offset * 4
    Memory[address,4] = Sd
```

Notes

Addressing mode

This is a special case of *Addressing Mode 5 - VFP load/store multiple* on page C5-24.

Conversions   In the programmer's model, FSTS does not perform any conversion on the value transferred. The source register Sd can contain either a single-precision floating-point number or an integer. The latter is typically obtained as the result of one of the floating-point-to-integer conversion instructions.

Implementations are free to hold the value in Sd in an internal format, provided that FSTS converts it to an external format and this conversion recovers the correct data, whether or not Sd contains a single-precision floating-point number or an integer.

C4.1.60 FSUBD

31		28	27	26	25	24	23	22	21	20	19		16	15		12	11	10	9	8	7	6	5	4	3		0
cond			1	1	1	0	0	0	1	1		Dn			Dd		1	0	1	1	0	1	0	0		Dm	
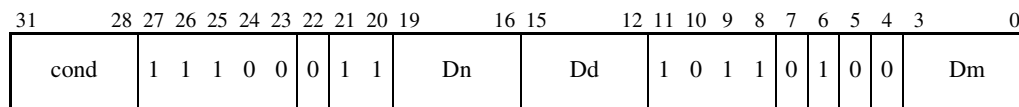
The FSUBD (Floating-point Subtract, Double-precision) instruction subtracts one double-precision register from another double-precision register and writes the result to a third double-precision register. It can also perform a vector version of this operation.

Syntax

```
FSUBD{<cond>}  <Dd>, <Dn>, <Dm>
```

where:

<cond>   Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-7. If <cond> is omitted, the AL (always) condition is used.

<Dd>   Specifies the destination register.

<Dn>   Specifies the register that contains the first operand for the subtraction.

<Dm>   Specifies the register that contains the second operand for the subtraction.

Architecture version

D variants only

Exceptions

Floating-point exceptions: Invalid Operation, Overflow, Inexact

Operation

```
if ConditionPassed(cond) then
    for i = 0 to vec_len-1
        Dd[i] = Dn[i] - Dm[i]
```

Notes

Vectors   When the LEN field of the FPSCR indicates scalar mode (vector length 1), FSUBD performs just one subtraction, and vec_len=1, Dd[0]=Dd, Dn[0]=Dn, and Dm[0]=Dm.

When the LEN field indicates a vector mode (vector length > 1), FSUBD might perform more than one subtraction. *Addressing Mode 2 - Double-precision vectors (non-monadic)* on page C5-8 describes how FSUBD encodes the registers it uses and how vec_len, Dd[i], Dn[i], and Dm[i] are determined.

Rounding   This is a fully-rounded subtraction. The rounding mode is determined by the FPSCR.

C4.1.61 FSUBS

31	28	27	26	25	24	23	22	21	20	19	16	15	12	11	10	9	8	7	6	5	4	3	0
cond		1	1	1	0	0	D	1	1	Fn		Fd		1	0	1	0	N	1	M	0	Fm	

The FSUBS (Floating-point Subtract, Single-precision) instruction subtracts one single-precision register from another single-precision register and writes the result to a third single-precision register. It can also perform a vector version of this operation.

Syntax

```
FSUBS{<cond>}  <Sd>, <Sn>, <Sm>
```

where:

<cond>	Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-7. If <cond> is omitted, the AL (always) condition is used.
<Sd>	Specifies the destination register. Its number is encoded as Fd (top 4 bits) and D (bottom bit).
<Sn>	Specifies the register that contains the first operand for the subtraction. The register number is encoded as Fn (top 4 bits) and N (bottom bit).
<Sm>	Specifies the register that contains the second operand for the subtraction. The register number is encoded as Fm (top 4 bits) and M (bottom bit).

Architecture version

All

Exceptions

Floating-point exceptions: Invalid Operation, Overflow, Inexact

Operation

```
if ConditionPassed(cond) then
    for i = 0 to vec_len-1
        Sd[i] = Sn[i] - Sm[i]
```

Notes

Vectors   When the LEN field of the FPSCR indicates scalar mode (vector length 1), `FSUBS` performs one subtraction, and `vec_len=1`, `Sd[0]=Sd`, `Sn[0]=Sn`, and `Sm[0]=Sm`.

When the LEN field indicates a vector mode (vector length > 1), `FSUBS` might perform more than one subtraction. *Addressing Mode 1 - Single-precision vectors (non-monadic)* on page C5-2 describes how `FSUBS` encodes the registers it uses and how `vec_len`, `Sd[i]`, `Sn[i]`, and `Sm[i]` are determined.

Rounding   The operation is a fully-rounded subtraction. Rounding mode is determined by the FPSCR.

C4.1.62 FTOSID

31		28	27	26	25	24	23	22	21	20	19		16	15		12	11	10	9	8	7	6	5	4	3		0
cond			1	1	1	0	1	D	1	1	1	1	0	1	Fd		1	0	1	1	Z	1	0	0	Dm		

The FTOSID (Floating-point Convert to Signed Integer from Double-precision) instruction converts a value held in a double-precision register to a signed integer and writes the result to a single-precision register. The integer value is normally then transferred to memory by a single-precision store instruction or to an ARM register by an FMRS instruction.

Syntax

```
FTOSI{Z}D{<cond>}  <Sd>, <Dm>
```

where:

Z	Sets the Z bit in the instruction to 1 and means that the operation uses the *round towards zero* rounding mode. If Z is not specified, the Z bit of the instruction is 0 and the operation uses the rounding mode specified by the FPSCR.
<cond>	Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-7. If <cond> is omitted, the AL (always) condition is used.
<Sd>	Specifies the destination register. Its number is encoded as Fd (top 4 bits) and D (bottom bit).
<Dm>	Specifies the source register.

Architecture version

D variants only

Exceptions

Floating-point exceptions: Invalid Operation, Inexact

Operation

```
if ConditionPassed(cond) then
    Sd = ConvertDoubleToSignedInteger(Dm)
```

Notes

Vectors   `FTOSID` always specifies a scalar operation, regardless of the LEN field of the FPSCR.

Out-of-range values

If the operand is $-\infty$ (minus infinity) or the result after rounding would be less than -2^{31}, an invalid operation exception is raised. If this exception is untrapped, the result is `0x80000000`.

If the operand is $+\infty$ (plus infinity) or the result after rounding would be greater than 2^{31-1}, an invalid operation exception is raised. If the exception is untrapped, the result is `0x7FFFFFFF`.

If the operand is a NaN, an invalid operation exception is raised. If this exception is untrapped, the result is `0x00000000`.

C4.1.63 FTOSIS

31		28	27	26	25	24	23	22	21	20	19			16	15			12	11	10	9	8	7	6	5	4	3		0
cond			1	1	1	0	1	D	1	1	1	1	0	1		Fd			1	0	1	0	Z	1	M	0		Fm	
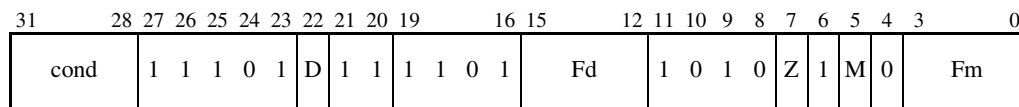
The FTOSIS (Floating-point Convert to Signed Integer from Single-precision) instruction converts a value held in a single-precision register to a signed integer and writes the result to a second single-precision register. The integer value is normally then transferred to memory by a single-precision store instruction or to an ARM register by an FMRS instruction.

Syntax

```
FTOSI{Z}S{<cond>}  <Sd>, <Sm>
```

where:

Z	Sets the Z bit in the instruction to 1 and means that the operation uses the *round towards zero* rounding mode. If Z is not specified, the Z bit of the instruction is 0 and the operation uses the rounding mode specified by the FPSCR.
\<cond\>	Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-7. If \<cond\> is omitted, the AL (always) condition is used.
\<Sd\>	Specifies the destination register. Its number is encoded as Fd (top 4 bits) and D (bottom bit).
\<Sm\>	Specifies the source register. Its number is encoded as Fm (top 4 bits) and M (bottom bit).

Architecture version

All

Exceptions

Floating-point exceptions: Invalid Operation, Inexact

Operation

```
if ConditionPassed(cond) then
    Sd = ConvertSingleToSignedInteger(Sm)
```

Notes

Vectors   FTOSIS always specifies a scalar operation, regardless of the LEN field of the FPSCR.

Out-of-range values

If the operand is $-\infty$ (minus infinity) or the result after rounding would be less than -2^{31}, an invalid operation exception is raised. If this exception is untrapped, the result is 0x80000000.

If the operand is $+\infty$ (plus infinity) or the result after rounding would be greater than $2^{31}-1$, an invalid operation exception is raised. If this exception is untrapped, the result is 0x7FFFFFFF.

If the operand is a NaN, an invalid operation exception is raised. If this exception is untrapped, the result is 0x00000000.

C4.1.64 FTOUID

31	28	27	26	25	24	23	22	21	20	19	16	15	12	11	10	9	8	7	6	5	4	3	0
cond		1	1	1	0	1	D	1	1	1 1 0 0		Fd		1	0	1	1	Z	1	0	0	Dm	
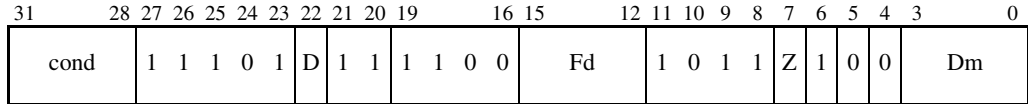
The FTOUID (Floating-point Convert to Unsigned Integer from Double-precision) instruction converts a value held in a double-precision register to an unsigned integer and writes the result to a single-precision register. The integer value is normally then transferred to memory by a single-precision store instruction or to an ARM register by an FMRS instruction.

Syntax

```
FTOUI{Z}D{<cond>}  <Sd>, <Dm>
```

where:

Z	Sets the Z bit in the instruction to 1 and means that the operation uses the *round towards zero* rounding mode. If Z is not specified, the Z bit of the instruction is 0 and the operation uses the rounding mode specified by the FPSCR.
<cond>	Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-7. If <cond> is omitted, the AL (always) condition is used.
<Sd>	Specifies the destination register. Its number is encoded as Fd (top 4 bits) and D (bottom bit).
<Dm>	Specifies the source register.

Architecture version

D variants only

Exceptions

Floating-point exceptions: Invalid Operation, Inexact

Operation

```
if ConditionPassed(cond) then
    Sd = ConvertDoubleToUnsignedInteger(Dm)
```

Notes

Vectors   FTOUID always specifies a scalar operation, regardless of the LEN field of the FPSCR.

Out-of-range values

If the operand is $-\infty$ (minus infinity) or the result after rounding would be less than 0, an invalid operation exception is raised. If this exception is untrapped, the result is 0x00000000.

If the operand is $+\infty$ (plus infinity) or the result after rounding would be greater than 2^{32-1}, an invalid operation exception is raised. If this exception is untrapped, the result is 0xFFFFFFFF.

If the operand is a NaN, an invalid operation exception is raised. If this exception is untrapped, the result is 0x00000000.

C4.1.65 FTOUIS

31				28	27	26	25	24	23	22	21	20	19				16	15				12	11	10	9	8	7	6	5	4	3			0
cond					1	1	1	0	1	D	1	1	1	1	0	0		Fd					1	0	1	0	Z	1	M	0	Fm			

The FTOUIS (Floating-point Convert to Unsigned Integer from Single-precision) instruction converts a value held in a single-precision register to an unsigned integer and writes the result to a second single-precision register. The integer value is normally then transferred to memory by a single-precision store instruction or to an ARM register by an FMRS instruction.

Syntax

FTOUI{Z}S{<cond>} <Sd>, <Sm>

where:

Z	Sets the Z bit in the instruction to 1 and means that the operation uses the *round towards zero* rounding mode. If Z is not specified, the Z bit of the instruction is 0 and the operation uses the rounding mode specified by the FPSCR.
<cond>	Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-7. If <cond> is omitted, the AL (always) condition is used.
<Sd>	Specifies the destination register. Its number is encoded as Fd (top 4 bits) and D (bottom bit).
<Sm>	Specifies the source register. Its number is encoded as Fm (top 4 bits) and M (bottom bit).

Architecture version

All

Exceptions

Floating-point exceptions: Invalid Operation, Inexact

Operation

```
if ConditionPassed(cond) then
    Sd = ConvertSingleToUnsignedInteger(Sm)
```

Notes

Vectors   FTOUIS always specifies a scalar operation, regardless of the LEN field of the FPSCR.

Out-of-range values

If the operand is −∞ (minus infinity) or the result after rounding would be less than 0, an invalid operation exception is raised. If this exception is untrapped, the result is 0x00000000.

If the operand is +∞ (plus infinity) or the result after rounding would be greater than 2^{32-1}, an invalid operation exception is raised. If this exception is untrapped, the result is 0xFFFFFFFF.

If the operand is a NaN, an invalid operation exception is raised. If this exception is untrapped, the result is 0x00000000.

C4.1.66 FUITOD

31	28	27	26	25	24	23	22	21	20	19		16	15		12	11	10	9	8	7	6	5	4	3		0
cond		1	1	1	0	1	0	1	1	1	0	0	0	Dd			1	0	1	1	0	1	M	0	Fm	

The FUITOD (Floating-point Convert Unsigned Integer to Double-precision) instruction converts an unsigned integer value held in a single-precision register to double-precision and writes the result to a double-precision register. The integer value will normally have been transferred from memory by a single-precision Load instruction or from an ARM register by an FMSR instruction.

Syntax

```
FUITOD{<cond>}  <Dd>, <Sm>
```

where:

<cond>   Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-7. If <cond> is omitted, the AL (always) condition is used.

<Dd>   Specifies the destination register.

<Sm>   Specifies the source register. The register number is encoded as Fm (top 4 bits) and M (bottom bit).

Architecture version

D variants only

Exceptions

None

Operation

```
if ConditionPassed(cond) then
    Dd = ConvertUnsignedIntegerToDouble(Sm)
```

Notes

Vectors   FUITOD always specifies a scalar operation, regardless of the LEN field of the FPSCR.

Zero   If Sm contains an integer zero, the result is a double-precision +0.0, not a double-precision -0.0.

C4.1.67 FUITOS

31		28	27	26	25	24	23	22	21	20	19			16	15			12	11	10	9	8	7	6	5	4	3		0
cond			1	1	1	0	1	D	1	1	1	0	0	0		Fd			1	0	1	0	0	1	M	0		Fm	

The FUITOS (Floating-point Convert Unsigned Integer to Single-precision) instruction converts an
unsigned integer value held in a single-precision register to single-precision and writes the result to a second
single-precision register. The integer value will normally have been transferred from memory by a
single-precision Load instruction or from an ARM register by an FMSR instruction.

Syntax

```
FUITOS{<cond>}  <Sd>, <Sm>
```

where:

<cond>   Is the condition under which the instruction is executed. The conditions are defined in *The
            condition field* on page A3-7. If <cond> is omitted, the AL (always) condition is used.

<Sd>   Specifies the destination register. The register number is encoded as Fd (top 4 bits) and D
            (bottom bit).

<Sm>   Specifies the source register. The register number is encoded as Fm (top 4 bits) and M
            (bottom bit).

Architecture version

All

Exceptions

Floating-point exception: Inexact

Operation

```
if ConditionPassed(cond) then
    Sd = ConvertUnsignedIntegerToSingle(Sm)
```

Notes

Vectors   FUITOS always specifies a scalar operation, regardless of the LEN field of the FPSCR.

Zero   If Sm contains an integer zero, the result is a single-precision +0.0, not a single-precision
               -0.0.

Rounding   Rounding is needed for some large operand values. The rounding mode is determined by the
               FPSCR.

Chapter C5
VFP Addressing Modes

This chapter describes the syntax and usage of each of the five VFP addressing modes. The chapter contains:

C5.1 Addressing Mode 1 - Single-precision vectors (non-monadic)

31		28	27	26	25	24	23	22	21	20	19		16	15		12	11	10	9	8	7	6	5	4	3		0
cond			1	1	1	0	Op	D	Op		Fn			Fd			1	0	1	0	N	Op	M	0	Fm		

When the vector length indicated by the FPSCR is greater than 1, the single-precision two-operand instructions `FADDS`, `FDIVS`, `FMULS`, `FNMULS`, and `FSUBS` can specify three different types of behavior:

- One arithmetic operation between two scalar values, yielding a scalar:

  ```
  ScalarA op ScalarB → ScalarD
  ```

  When this case is selected (see *Scalar operations* on page C5-5), it causes just one operation to be performed, overriding the vector length specified in the FPSCR. This allows scalar operations and vector operations to be mixed without the need to reprogram the FPSCR between them.

- A set of `N` arithmetic operations, where `N` is the vector length specified in the FPSCR, with the first operand scanning through a vector, the second operand remaining constant and the destination scanning through a vector:

  ```
  VectorA[0] op ScalarB → VectorD[0]
  VectorA[1] op ScalarB → VectorD[1]
  ...
  VectorA[N-1] op ScalarB → VectorD[N-1]
  ```

  This can be abbreviated to:

  ```
  VectorA op ScalarB → VectorD
  ```

- A set of `N` arithmetic operations, where `N` is the vector length specified in the FPSCR, with both operands and the destination scanning through vectors:

  ```
  VectorA[0] op VectorB[0] → VectorD[0]
  VectorA[1] op VectorB[1] → VectorD[1]
  ...
  VectorA[N-1] op VectorB[N-1] → VectorD[N-1]
  ```

  This can be abbreviated to:

  ```
  VectorA op VectorB → VectorD
  ```

The single-precision three-operand instructions `FMACS`, `FMSCS`, `FNMACS` and `FNMSCS` each use the same register for their addition/subtraction operand as for their destination. So they have three forms corresponding to the above three:

- A pure scalar form:

  ```
  ± (ScalarA * ScalarB) ± ScalarD → ScalarD
  ```

- A form in which the second multiplication operand is a scalar and everything else scans through vectors:

  ```
  ± (VectorA[0]   * ScalarB) ± VectorD[0] → VectorD[0]
  ± (VectorA[1]   * ScalarB) ± VectorD[1] → VectorD[1]
  ...
  ± (VectorA[N-1] * ScalarB) ± VectorD[N-1] → VectorD[N-1]
  ```

  This can be abbreviated to:

  ```
  ± (VectorA * ScalarB) ± VectorD → VectorD
  ```

- A form in which everything scans through a vector:

```
± (VectorA[0] * VectorB[0]) ± VectorD[0]   → VectorD[0]
± (VectorA[1] * VectorB[1]) ± VectorD[1]   → VectorD[1]
...
± (VectorA[N-1] * VectorB[N-1]) ± VectorD[N-1] → VectorD[N-1]
```

This can be abbreviated to:

```
± (VectorA * VectorB) ± VectorD → VectorD
```

C5.1.1   Register banks

To allow these various forms to be specified, the set of 32 single-precision registers is split into four banks, each of eight registers. The form used by an instruction depends on which operands are in the first bank. The general principle behind the rules is that the first bank must be used to hold scalar operands while the other banks are used to hold vector operands. All destination register writes and many source register reads adhere to this principle, but some source register reads can result in scalar access to vector elements or vector accesses to groups of scalars.

A vector operand consists of 2-8 registers from a single bank, with the number of registers being specified by the vector length field of the FPSCR (see *Vector length/stride control* on page C2-24). The register number in the instruction specifies the register that contains the first element of the vector. Each successive element of the vector is formed by incrementing the register number by the value specified by the vector stride field of the FPSCR. If this causes the register number to overflow the top of the register bank, the register number wraps around to the bottom of the bank, as shown in Figure C5-1.

Figure C5-1 Single-precision register banks

C5.1.2 Operation

The following sections describe each of the three possible forms of the addressing mode:

- *Scalar operations* on page C5-5
- *Mixed vector/scalar operations* on page C5-6
- *Vector operations* on page C5-7.

In each case, the following values are generated:

vec_len   The number of individual operations specified by the instruction.

Sd[0] ... Sd[vec_len-1]

Destination registers of the individual operations.

Sn[0] ... Sn[vec_len-1]

First source registers of the individual operations.

Sm[0] ... Sm[vec_len-1]

Second source registers of the individual operations.

In all cases, the registers specified by the instruction are determined by concatenating the Fd, Fn and Fm fields of the instruction with the D, N and M bits respectively:

```
d_num = (Fd << 1) | D
n_num = (Fn << 1) | N
m_num = (Fm << 1) | M
```

These register numbers are then broken up into bank numbers and indices within the banks as follows:

```
d_bank  = d_num[4:3]
d_index = d_num[2:0]

n_bank  = n_num[4:3]
n_index = n_num[2:0]

m_bank  = m_num[4:3]
m_index = m_num[2:0]
```

——— **Note** ———

The case where the FPSCR specifies a vector length of 1 is not in fact a special case, because the rules for all three forms of the addressing mode simplify to the following when the vector length is 1:

```
vec_len = 1
Sd[0] = d_num
Sn[0] = n_num
Sm[0] = m_num
```

C5.1.3 Scalar operations

If the destination register lies in the first bank of eight registers, the instruction specifies a scalar operation:

```
if d_bank == 0 then
    vec_len = 1
    Sd[0] = d_num
    Sn[0] = n_num
    Sm[0] = m_num
```

Note

Source operands   The source operands are always scalars, regardless of which bank they are in. This allows individual elements of vectors to be used as scalars.

Copyright © 1996–2003 ARM Limited. All rights reserved.
Confidential - Draft

C5.1.4   Mixed vector/scalar operations

If the destination register specified in the instruction does not lie in the first bank of eight registers, but the second source register does, then the destination register and first source register specify vectors and the second source register specifies a scalar:

```
if d_bank != 0 and m_bank == 0 then
    vec_len = vector length specified by FPSCR
    for i = 0 to vec_len-1
        Sd[i] = (d_bank << 3) | d_index
        Sn[i] = (n_bank << 3) | n_index
        Sm[i] = m_num
        d_index = d_index + (vector stride specified by FPSCR)
        if d_index > 7 then
            d_index = d_index - 8
        n_index = n_index + (vector stride specified by FPSCR)
        if n_index > 7 then
            n_index = n_index - 8
```

Notes

First source operand

The first operand is always a vector, regardless of which bank it is in. This allows a set of consecutive registers in the first bank to be treated as a vector.

Vector wrap-around

A vector operand must not wrap around so that it re-uses its first element. Otherwise, the results of the instruction are UNPREDICTABLE. When the FPSCR specifies a vector stride of 1, this is not a restriction, because the vector length is at most 8. When the FPSCR specifies a vector stride of 2, it implies that the vector length must be at most 4.

Operand overlap

If two operands overlap, they must be identical both in terms of which registers are accessed and the order in which they are accessed. Otherwise, the results of the instruction are UNPREDICTABLE. This implies that:

- If the set of register numbers generated in Sd[i] overlaps the set of register numbers generated in Sn[i], then d_num and n_num must be identical.

- If the set of register numbers generated in Sn[i] includes m_num, the vector length must be 1.

It is impossible for the set of register numbers generated in Sd[i] to include m_num, because they lie in different banks.

C5.1.5   Vector operations

If neither the destination register nor the second source register lies in the first bank of eight registers, then all register operands specify vectors:

```
if d_bank != 0 and m_bank != 0 then
    vec_len = vector length specified by FPSCR
    for i = 0 to vec_len-1
        Sd[i] = (d_bank << 3) | d_index
        Sn[i] = (n_bank << 3) | n_index
        Sm[i] = (m_bank << 3) | m_index
        d_index = d_index + (vector stride specified by FPSCR)
        if d_index > 7 then
            d_index = d_index - 8
        n_index = n_index + (vector stride specified by FPSCR)
        if n_index > 7 then
            n_index = n_index - 8
        m_index = m_index + (vector stride specified by FPSCR)
        if m_index > 7 then
            m_index = m_index - 8
```

Notes

Vector wrap-around   A vector operand must not wrap around so that it re-uses its first element. Otherwise, the results of the instruction are UNPREDICTABLE. When the FPSCR specifies a vector stride of 1, this is not a restriction, since the vector length is at most 8. When the FPSCR specifies a vector stride of 2, it implies that the vector length must be at most 4.

Operand overlap   If two operands overlap, they must be identical both in terms of which registers are accessed and the order in which they are accessed. Otherwise, the results of the instruction are UNPREDICTABLE. This implies that:

- If the set of register numbers generated in Sd[i] overlaps the set of register numbers generated in Sn[i], then d_num and n_num must be identical.

- If the set of register numbers generated in Sd[i] overlaps the set of register numbers generated in Sm[i], then d_num and m_num must be identical.

- If the set of register numbers generated in Sn[i] overlaps the set of register numbers generated in Sm[i], then n_num and m_num must be identical.

C5.2   Addressing Mode 2 - Double-precision vectors (non-monadic)

31				28	27	26	25	24	23	22	21	20	19				16	15				12	11	10	9	8	7	6	5	4	3				0
cond					1	1	1	0	Op	0	Op		Dn					Dd					1	0	1	1	0	Op	0	0	Dm				
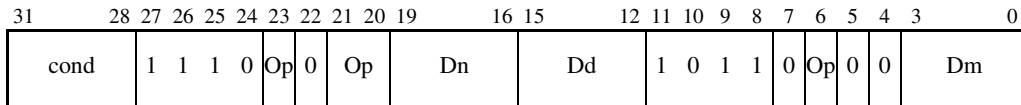
When the vector length indicated by the FPSCR is greater than 1, the double-precision two-operand instructions FADDD, FDIVD, FMULD, FNMULD, and FSUBD can specify three different types of behavior:

- One arithmetic operation between two scalar values, yielding a scalar:

  ```
  ScalarA op ScalarB → ScalarD
  ```

  When this case is selected (see *Scalar operations* on page C5-11), it causes just one operation to be performed, overriding the vector length specified in the FPSCR. This allows scalar operations and vector operations to be mixed without the need to reprogram the FPSCR between them.

- A set of N arithmetic operations, where N is the vector length specified in the FPSCR, with the first operand scanning through a vector, the second operand remaining constant and the destination scanning through a vector:

  ```
  VectorA[0] op ScalarB → VectorD[0]
  VectorA[1] op ScalarB → VectorD[1]
  ...
  VectorA[N-1] op ScalarB → VectorD[N-1]
  ```

  This can be abbreviated to:

  ```
  VectorA op ScalarB → VectorD
  ```

- A set of N arithmetic operations, where N is the vector length specified in the FPSCR, with both operands and the destination scanning through vectors:

  ```
  VectorA[0] op VectorB[0] → VectorD[0]
  VectorA[1] op VectorB[1] → VectorD[1]
  ...
  VectorA[N-1] op VectorB[N-1] → VectorD[N-1]
  ```

  This can be abbreviated to:

  ```
  VectorA op VectorB → VectorD
  ```

The double-precision three-operand instructions FMACD, FMSCD, FNMACD and FNMSCD each use the same register for their addition/subtraction operand as for their destination. So they have three forms corresponding to the above three:

- A pure scalar form:

  ```
  ± (ScalarA * ScalarB) ± ScalarD → ScalarD
  ```

- A form in which the second multiplication operand is a scalar and everything else scans through vectors:

  ```
  ± (VectorA[0] * ScalarB) ± VectorD[0] → VectorD[0]
  ± (VectorA[1] * ScalarB) ± VectorD[1] → VectorD[1]
  ...
  ± (VectorA[N-1] * ScalarB) ± VectorD[N-1] → VectorD[N-1]
  ```

  This can be abbreviated to:

  ```
  ± (VectorA * ScalarB) ± VectorD → VectorD
  ```

• A form in which everything scans through a vector:

```
± (VectorA[0]  * VectorB[0]) ± VectorD[0]  → VectorD[0]
± (VectorA[1]  * VectorB[1]) ± VectorD[1]  → VectorD[1]
...
± (VectorA[N-1] * VectorB[N-1]) ± VectorD[N-1] → VectorD[N-1]
```

This can be abbreviated to:

```
± (VectorA * VectorB) ± VectorD → VectorD
```

C5.2.1 Register banks

To allow these various forms to be specified, the set of 16 double-precision registers is split into four banks, each of four registers. The form used by an instruction depends on which operands are in the first bank. The general principle behind the rules is that the first bank must be used to hold scalar operands while the other banks are used to hold vector operands. All destination register writes and many source register reads adhere to this principle, but some source register reads can result in scalar access to vector elements or vector accesses to groups of scalars.

A vector operand consists of 2-4 registers from a single bank, with the number of registers being specified by the vector length field of the FPSCR (see *Vector length/stride control* on page C2-24). The register number in the instruction specifies the register that contains the first element of the vector. Each successive element of the vector is formed by incrementing the register number by the value specified by the vector stride field of the FPSCR. If this causes the register number to overflow the top of the register bank, the register number wraps around to the bottom of the bank, as shown in Figure C5-2.

Figure C5-2 Double-precision register banks

C5.2.2 Operation

The following pages describe each of the three possible forms of the addressing mode:

In each case, the following values are generated:

`vec_len`   The number of individual operations specified by the instruction.

`Dd[0] ... Dd[vec_len-1]`

     Destination registers of the individual operations.

`Dn[0] ... Dn[vec_len-1]`

     First source registers of the individual operations.

`Dm[0] ... Dm[vec_len-1]`

     Second source registers of the individual operations.

The register numbers specified in the instruction are broken up into bank numbers and indices within the banks as follows:

```
d_bank  = Dd[3:2]
d_index = Dd[1:0]

n_bank  = dn[3:2]
n_index = Dn[1:0]

m_bank  = Dm[3:2]
m_index = Dm[1:0]
```

——— **Note** ———

The case where the FPSCR specifies a vector length of 1 is not in fact a special case, since the rules for all three forms of the addressing mode simplify to the following when the vector length is 1:

```
vec_len = 1
Dd[0] = Dd
Dn[0] = Dn
Dm[0] = Dm
```

C5.2.3   Scalar operations

If the destination register lies in the first bank of four registers, the instruction specifies a scalar operation:

```
if d_bank == 0 then
    vec_len = 1
    Dd[0] = Dd
    Dn[0] = Dn
    Dm[0] = Dm
```

Notes

Source operands   The source operands are always scalars, regardless of which bank they are in. This allows individual elements of vectors to be used as scalars.

C5.2.4 Mixed vector/scalar operations

If the destination register specified in the instruction does not lie in the first bank of four registers, but the second source register does, then the destination register and first source register specify vectors and the second source register specifies a scalar:

```
if d_bank != 0 and m_bank == 0 then
    vec_len = vector length specified by FPSCR
    for i = 0 to vec_len-1
        Dd[i] = (d_bank << 2) | d_index
        Dn[i] = (n_bank << 2) | n_index
        Dm[i] = Dm
        d_index = d_index + (vector stride specified by FPSCR)
        if d_index > 3 then
            d_index = d_index - 4
        n_index = n_index + (vector stride specified by FPSCR)
        if n_index > 3 then
            n_index = n_index - 4
```

Notes

First source operand The first operand is always a vector, regardless of which bank it is in. This allows a set of consecutive registers in the first bank to be treated as a vector.

Vector wrap-around A vector operand must not wrap around so that it re-uses its first element. Otherwise, the results of the instruction are UNPREDICTABLE. When the FPSCR specifies a vector stride of 1, this implies that the vector length must be at most 4. When the FPSCR specifies a vector stride of 2, it implies that the vector length must be at most 2.

Operand overlap If two operands overlap, they must be identical both in terms of which registers are accessed and the order in which they are accessed. Otherwise, the results of the instruction are UNPREDICTABLE. This implies that:

- If the set of register numbers generated in Dd[i] overlaps the set of register numbers generated in Dn[i], then Dd and Dn must be identical.

- If the set of register numbers generated in Dn[i] includes Dm, then the vector length must be 1.

It is impossible for the set of register numbers generated in Dd[i] to include Dm, because they lie in different banks.

C5.2.5   Vector operations

If neither the destination register nor the second source register lies in the first bank of four registers, then all register operands specify vectors:

```
if d_bank != 0 and m_bank != 0 then
    vec_len = vector length specified by FPSCR
    for i = 0 to vec_len-1
        Dd[i] = (d_bank << 2) | d_index
        Dn[i] = (n_bank << 2) | n_index
        Dm[i] = (m_bank << 2) | m_index
        d_index = d_index + (vector stride specified by FPSCR)
        if d_index > 3 then
            d_index = d_index - 4
        n_index = n_index + (vector stride specified by FPSCR)
        if n_index > 3 then
            n_index = n_index - 4
        m_index = m_index + (vector stride specified by FPSCR)
        if m_index > 3 then
            m_index = m_index - 4
```

Notes

Vector wrap-around   A vector operand must not wrap around so that it re-uses its first element. Otherwise, the results of the instruction are UNPREDICTABLE. When the FPSCR specifies a vector stride of 1, this implies that the vector length must be at most 4. When the FPSCR specifies a vector stride of 2, it implies that the vector length must be at most 2.

Operand overlap   If two operands overlap, they must be identical both in terms of which registers are accessed and the order in which they are accessed. Otherwise, the results of the instruction are UNPREDICTABLE. This implies that:

- If the set of register numbers generated in Dd[i] overlaps the set of register numbers generated in Dn[i], then Dd and Dn must be identical.

- If the set of register numbers generated in Dd[i] overlaps the set of register numbers generated in Dm[i], then Dd and Dm must be identical.

- If the set of register numbers generated in Dn[i] overlaps the set of register numbers generated in Dm[i], then Dn and Dm must be identical.

Copyright © 1996–2003 ARM Limited. All rights reserved.
Confidential - Draft

C5.3   Addressing Mode 3 - Single-precision vectors (monadic)

31			28	27	26	25	24	23	22	21	20	19			16	15			12	11	10	9	8	7	6	5	4	3			0
cond				1	1	1	0	1	D	1	1	Op				Fd				1	0	1	0	Op	1	M	0	Fm			
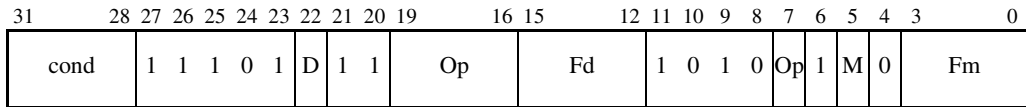
When the vector length indicated by the FPSCR is greater than 1, the single-precision one-operand instructions FABSS, FCPYS, FNEGS, and FSQRTS can specify three different types of behavior:

- An operation on a scalar value, yielding a scalar:

  Op(ScalarB) → ScalarD

  When this case is selected (see *Scalar-to-scalar operations* on page C5-16), it causes just one operation to be performed, overriding the vector length specified in the FPSCR. This allows scalar operations and vector operations to be mixed without the need to reprogram the FPSCR between them.

- An operation on a scalar value, whose result is written to each of the N elements of a vector, where N is the vector length specified in the FPSCR:

  Op(ScalarB) → VectorD[0]
  Op(ScalarB) → VectorD[1]
  ...
  Op(ScalarB) → VectorD[N-1]

  This can be abbreviated to:

  Op(ScalarB) → VectorD

- A set of N operations, where N is the vector length specified in the FPSCR, with both the operand and the destination scanning through vectors:

  Op(VectorB[0]) → VectorD[0]
  Op(VectorB[1]) → VectorD[1]
  ...
  Op(VectorB[N-1]) → VectorD[N-1]

  This can be abbreviated to:

  Op(VectorB) → VectorD

To allow these various forms to be specified, the set of 32 single-precision registers is split into four banks, each of eight registers. For a description of this, see *Register banks* on page C5-3.

ARM DDI 0100F-02

C5.3.1 Operation

The following pages describe each of the three possible forms of the addressing mode:

- *Scalar-to-scalar operations* on page C5-16
- *Scalar-to-vector operations* on page C5-17
- *Vector-to-vector operations* on page C5-18.

In each case, the following values are generated:

vec_len   The number of individual operations specified by the instruction.

Sd[0] ... Sd[vec_len-1]

          Destination registers of the individual operations.

Sm[0] ... Sm[vec_len-1]

          Source registers of the individual operations.

In all cases, the registers specified by the instruction are determined by concatenating the Fd and Fm fields of the instruction with the D and M bits respectively:

```
d_num = (Fd << 1) | D
m_num = (Fm << 1) | M
```

These register numbers are then broken up into bank numbers and indices within the banks as follows:

```
d_bank  = d_num[4:3]
d_index = d_num[2:0]

m_bank  = m_num[4:3]
m_index = m_num[2:0]
```

———— **Note** ————

The case where the FPSCR specifies a vector length of 1 is not in fact a special case, since the rules for all three forms of the addressing mode simplify to the following when the vector length is 1:

```
vec_len = 1
Sd[0] = d_num
Sm[0] = m_num
```

————————————————

C5.3.2   Scalar-to-scalar operations

If the destination register lies in the first bank of eight registers, the instruction specifies a scalar operation:

```
if d_bank == 0 then
    vec_len = 1
    Sd[0] = d_num
    Sm[0] = m_num
```

Notes

Source operands   The source operand is always a scalar, regardless of which bank it lies in. This allows individual elements of vectors to be used as scalars.

C5.3.3   Scalar-to-vector operations

If the destination register specified in the instruction does not lie in the first bank of eight registers, but the source register does, then the destination register specifies a vector and the source register specifies a scalar:

```
if d_bank != 0 and m_bank == 0 then
    vec_len = vector length specified by FPSCR
    for i = 0 to vec_len-1
        Sd[i] = (d_bank << 3) | d_index
        Sm[i] = m_num
        d_index = d_index + (vector stride specified by FPSCR)
        if d_index > 7 then
            d_index = d_index - 8
```

Notes

Vector wrap-around   A vector operand must not wrap around so that it re-uses its first element. Otherwise, the results of the instruction are UNPREDICTABLE. When the FPSCR specifies a vector stride of 1, this is not a restriction, because the vector length is at most 8. When the FPSCR specifies a vector stride of 2, it implies that the vector length must be at most 4.

Operand overlap   If the source and destination overlap, they must be identical both in terms of which registers are accessed and the order in which they are accessed. This implies that if the set of register numbers generated in Sn[i] includes m_num, the vector length must be 1.

Copyright © 1996–2003 ARM Limited. All rights reserved.
Confidential - Draft

C5.3.4 Vector-to-vector operations

If neither the destination register nor the source register lies in the first bank of eight registers, then both register operands specify vectors:

```
if d_bank != 0 and m_bank != 0 then
    vec_len = vector length specified by FPSCR
    for i = 0 to vec_len-1
        Sd[i] = (d_bank << 3) | d_index
        Sm[i] = (m_bank << 3) | m_index
        d_index = d_index + (vector stride specified by FPSCR)
        if d_index > 7 then
            d_index = d_index - 8
        m_index = m_index + (vector stride specified by FPSCR)
        if m_index > 7 then
            m_index = m_index - 8
```

Notes

Vector wrap-around A vector operand must not wrap around so that it re-uses its first element. Otherwise, the results of the instruction are UNPREDICTABLE. When the FPSCR specifies a vector stride of 1, this is not a restriction, since the vector length is at most 8. When the FPSCR specifies a vector stride of 2, it implies that the vector length must be at most 4.

Operand overlap If the source and destination overlap, they must be identical both in terms of which registers are accessed and the order in which they are accessed. Otherwise, the results of the instruction are UNPREDICTABLE. This implies that if the set of register numbers generated in Sd[i] overlaps the set of register numbers generated in Sm[i], d_num and m_num must be identical.

C5.4   Addressing Mode 4 - Double-precision vectors (monadic)

31		28	27	26	25	24	23	22	21	20	19		16	15		12	11	10	9	8	7	6	5	4	3		0
cond			1	1	1	0	1	0	1	1		Op			Dd		1	0	1	1	Op	1	0	0		Dm	
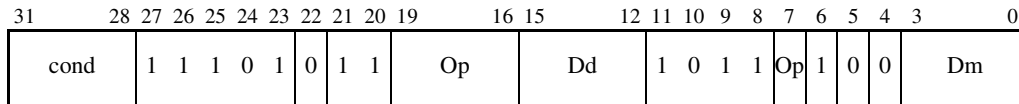
When the vector length indicated by the FPSCR is greater than 1, the double-precision one-operand instructions FABSD, FCPYD, FNEGD, and FSQRTD can specify three different types of behavior:

• An operation on a scalar value, yielding a scalar:

```
Op(ScalarB)  -->  ScalarD
```

When this case is selected (see *Scalar-to-scalar operations* on page C5-21), it causes just one operation to be performed, overriding the vector length specified in the FPSCR. This allows scalar operations and vector operations to be mixed without the need to reprogram the FPSCR between them.

• An operation on a scalar value, whose result is written to each of the N elements of a vector, where N is the vector length specified in the FPSCR:

```
Op(ScalarB)  -->  VectorD[0]
Op(ScalarB)  -->  VectorD[1]
...
Op(ScalarB)  -->  VectorD[N-1]
```

This can be abbreviated to:

```
Op(ScalarB)  -->  VectorD
```

• A set of N operations, where N is the vector length specified in the FPSCR, with both the operand and the destination scanning through vectors:

```
Op(VectorB[0])    -->  VectorD[0]
Op(VectorB[1])    -->  VectorD[1]
...
Op(VectorB[N-1])  -->  VectorD[N-1]
```

This can be abbreviated to:

```
Op(VectorB)  -->  VectorD
```

To allow these various forms to be specified, the set of 16 double-precision registers is split into four banks, each of four registers. For a description of this, see *Register banks* on page C5-9.

C5.4.1 Operation

The following pages describe each of the three possible forms of the addressing mode:

- *Scalar-to-scalar operations* on page C5-21
- *Scalar-to-vector operations* on page C5-22
- *Vector-to-vector operations* on page C5-23.

In each case, the following values are generated:

vec_len   The number of individual operations specified by the instruction.

Dd[0] ... Dd[vec_len-1]

Destination registers of the individual operations.

Dm[0] ... Dm[vec_len-1]

Source registers of the individual operations.

The register numbers specified in the instruction are broken up into bank numbers and indices within the banks as follows:

```
d_bank  = Dd[3:2]
d_index = Dd[1:0]

m_bank  = Dm[3:2]
m_index = Dm[1:0]
```

—— **Note** ——

The case where the FPSCR specifies a vector length of 1 is not in fact a special case, since the rules for all three forms of the addressing mode simplify to the following when the vector length is 1:

```
vec_len = 1
Dd[0] = Dd
Dm[0] = Dm
```

C5.4.2 Scalar-to-scalar operations

If the destination register lies in the first bank of four registers, the instruction specifies a scalar operation:

```
if d_bank == 0 then
    vec_len = 1
    Dd[0] = Dd
    Dm[0] = Dm
```

Notes

Source operands   The source operand is always a scalar, regardless of which bank it lies in. This allows individual elements of vectors to be used as scalars.

C5.4.3 Scalar-to-vector operations

If the destination register specified in the instruction does not lie in the first bank of four registers, but the source register does, then the destination register specifies a vector and the source register specifies a scalar:

```
if d_bank != 0 and m_bank == 0 then
    vec_len = vector length specified by FPSCR
    for i = 0 to vec_len-1
        Dd[i] = (d_bank << 2) | d_index
        Dm[i] = m_num
        d_index = d_index + (vector stride specified by FPSCR)
        if d_index > 3 then
            d_index = d_index - 4
```

Notes

Vector wrap-around A vector operand must not wrap around so that it re-uses its first element. Otherwise, the results of the instruction are UNPREDICTABLE. When the FPSCR specifies a vector stride of 1, this implies that the vector length must be at most 4. When the FPSCR specifies a vector stride of 2, it implies that the vector length must be at most 2.

Operand overlap If the source and destination overlap, they must be identical both in terms of which registers are accessed and the order in which they are accessed. This implies that if the set of register numbers generated in Dn[i] includes Dm, the vector length must be 1.

C5.4.4   Vector-to-vector operations

If neither the destination register nor the source register lies in the first bank of four registers, then both register operands specify vectors:

```
if d_bank != 0 and m_bank != 0 then
    vec_len = vector length specified by FPSCR
    for i = 0 to vec_len-1
        Dd[i] = (d_bank << 2) | d_index
        Dm[i] = (m_bank << 2) | m_index
        d_index = d_index + (vector stride specified by FPSCR)
        if d_index > 3 then
            d_index = d_index - 4
        m_index = m_index + (vector stride specified by FPSCR)
        if m_index > 3 then
            m_index = m_index - 4
```

Notes

Vector wrap-around   A vector operand must not wrap around so that it re-uses its first element. Otherwise, the results of the instruction are UNPREDICTABLE. When the FPSCR specifies a vector stride of 1, this implies that the vector length must be at most 4. When the FPSCR specifies a vector stride of 2, it implies that the vector length must be at most 2.

Operand overlap   If the source and destination overlap, they must be identical both in terms of which registers are accessed and the order in which they are accessed. Otherwise, the results of the instruction are UNPREDICTABLE. This implies that if the set of register numbers generated in Dd[i] overlaps the set of register numbers generated in Dm[i], then Dd and Dm must be identical.

C5.5   Addressing Mode 5 - VFP load/store multiple

31 28	27 26 25	24	23	22	21	20 19 16	15 12	11 8	7 0
cond	1 1 1	P	U	D	W	L Rn	Fd	cp_num	offset
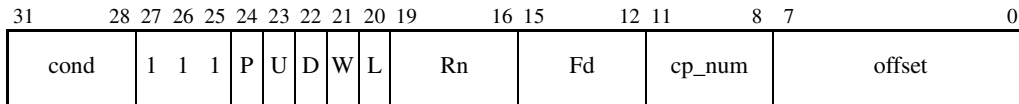
The VFP load multiple instructions (FLDMD, FLDMS, FLDMX) are examples of ARM LDC instructions, whose addressing modes are described in *Addressing Mode 5 - Load and Store Coprocessor* on page A5-60. Similarly, the VFP store multiple instructions (FSTMD, FSTMS, FSTMX) are examples of ARM STC instructions, which have the same addressing modes. However, the full range of LDC/STC addressing modes is not available for the VFP load multiple and store multiple instructions. This is partly because the FLDD, FLDS, FSTD and FSTS instructions use some of the options, and partly because the 8_bit_offset field in the LDC/STC instruction is used for additional purposes in the VFP instructions.

This section gives details of the LDC/STC addressing modes that are allowed for the VFP load multiple and store multiple instructions, and the assembler syntax for each option.

C5.5.1   Summary

Whether an LDC/STC addressing mode is allowed for the VFP load multiple and store multiple instructions can be determined by looking at the P, U and W bits of the instruction. Table C5-1 shows details of this.

Table C5-1 VFP load/store addressing modes

P	U	W	Instructions	Mode
0	0	0	UNDEFINED	See Note
0	0	1	UNDEFINED	See Note
0	1	0	FLDMD, FLDMS, FLDMX, FSTMD, FSTMS, FSTMX	Unindexed
0	1	1	FLDMD, FLDMS, FLDMX, FSTMD, FSTMS, FSTMX	Increment
1	0	0	FLDD, FLDS, FSTD, FSTS	(Negative offset)
1	0	1	FLDMD, FLDMS, FLDMX, FSTMD, FSTMS, FSTMX	Decrement
1	1	0	FLDD, FLDS, FSTD, FSTS	(Positive offset)
1	1	1	UNDEFINED	See following note

—— **Note** ——

For a hardware coprocessor implementation of the VFP instruction set, the UNDEFINED entries in Table C5-1 mean the coprocessor does not respond to the instruction, which make the ARM's Undefined Instruction exception occur (see *Undefined Instruction exception* on page A2-18).

For a software implementation, the UNDEFINED entries mean that such instructions must be passed to the system's normal mechanism for dealing with non-coprocessor undefined instructions. The exact details of this are system-dependent.

C5.5.2   VFP load/store multiple - Unindexed

31	28	27	26	25	24	23	22	21	20	19		16	15		12	11		8	7		0
cond		1	1	0	0	1	D	0	L	Rn			Fd			cp_num			offset		

This addressing mode is for VFP load multiple and store multiple instructions, and forms a range of addresses. The first address formed is the start_address, and is the value of the base register Rn. Subsequent addresses are formed by incrementing the previous address by four.

• For the FLDMS and FSTMS instructions, the offset in the instruction is equal to the number of single-precision registers to be transferred. One address is generated for each register, so the end_address is four less than the value of the base register Rn plus four times the offset.

• For the FLDMD and FSTMD instructions, the offset in the instruction is equal to twice the number of double-precision registers to be transferred. Two addresses are generated for each register, so the end_address is four less than the value of the base register Rn plus four times the offset.

• For the FLDMX and FSTMX instructions, the offset in the instruction is one more than twice the number of double-precision registers to be transferred.

The number of addresses generated is at most equal to the offset, but can be a smaller number (decided by the implementor) provided the FLDMX and FSTMX instructions function correctly (see *FLDMX* on page C4-435 and *FSTMX* on page C4-108). Accordingly, the end_address is the value of the base register Rn plus four times the offset, minus an IMPLEMENTATION DEFINED amount which is at least four.

Instruction syntax

```
<opcode>IA<precision>{<cond>}  <Rn>, <registers>
```

where:

<opcode>	Is FLDM or FSTM, and controls the value of the L bit.
<precision>	Is D, S or X, and controls the values of cp_num and offset[0].
<cond>	Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-7. If <cond> is omitted, the AL (always) condition is used.
<Rn>	Specifies the base register. If R15 is specified for <Rn>, the value used is the address of the instruction plus 8.
<registers>	Specifies the list of registers loaded or stored by the instruction. See the individual instructions for details of which registers are specified and how Fd, D and offset are set in the instruction.

Architecture version

All

Operation

```
if (offset[0] == 1) and (cp_num == 0b1011) then          /* FLDMX or FSTMX */
    word_count = IMPLEMENTATION DEFINED value (<= offset)
else                                                     /* Others */
    word_count = offset
start_address = Rn
end_address = start_address + 4 * word_count - 4
```

Usage

For FLDMD, FLDMS, FSTMD and FSTMS, this addressing mode is typically used to load or store a short vector. For example, to load a graphics point consisting of four single-precision coordinates into s8-s11, the following code might be used:

```
ADR      Rn, Point
FLDMIAS  Rn, {s8-s11}
```

For FLDMX and FSTMX, this addressing mode is typically used as part of loading and saving the VFP state in process swap code, in sequences like:

```
; Assume Rp points to the process block
ADD       Rn, Rp, #Offset_to_VFP_register_dump
FSTMIAX        Rn, {d0-d15}
```

Notes

Offset restrictions   The offset value must be at least 1 and at most 33. If the offset is 0 or greater than 33, the instruction is always UNPREDICTABLE. Each instruction also imposes further restrictions on the offset, depending on the values of Fd and D. See the individual instruction descriptions for details of these.

Copyright © 1996–2003 ARM Limited. All rights reserved.
Confidential - Draft

C5.5.3   VFP load/store multiple - Increment

31	28	27	26	25	24	23	22	21	20	19	16	15	12	11	8	7	0
cond		1	1	0	0	1	D	1	L	Rn		Fd		cp_num		offset	

This addressing mode is for VFP load multiple and store multiple instructions, and forms a range of addresses. The first address formed is the start_address, and is the value of the base register Rn. Subsequent addresses are formed by incrementing the previous address by four.

- For the `FLDMS` and `FSTMS` instructions, the offset in the instruction is equal to the number of single-precision registers to be transferred. One address is generated for each register, so the end_address is four less than the value of the base register Rn plus four times the offset.

- For the `FLDMD` and `FSTMD` instructions, the offset in the instruction is equal to twice the number of double-precision registers to be transferred. Two addresses are generated for each register, so the end_address is four less than the value of the base register Rn plus four times the offset.

- For the `FLDMX` and `FSTMX` instructions, the offset in the instruction is one more than twice the number of double-precision registers to be transferred.

The number of addresses generated is at most equal to the offset, but can be a smaller number (decided by the implementor) provided the `FLDMX` and `FSTMX` instructions function correctly (see *FLDMX* on page C4-43 and *FSTMX* on page C4-108). Accordingly, the end_address is the value of the base register Rn plus four times the offset, minus an IMPLEMENTATION DEFINED amount which is at least four.

For all instructions, if the condition specified in the instruction matches the condition code status (see *The condition field* on page A3-7), Rn is incremented by four times the offset specified in the instruction.

Instruction syntax

```
<opcode>IA<precision>{<cond>}  <Rn>!, <registers>
```

where:

`<opcode>`	Is `FLDM` or `FSTM`, and controls the value of the L bit.
`<precision>`	Is D, S or X, and controls the values of cp_num and offset[0].
`<cond>`	Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-7. If `<cond>` is omitted, the AL (always) condition is used.
`<Rn>`	Is the base register. If R15 is specified for `<Rn>`, the instruction is UNPREDICTABLE.
`!`	Indicates the base register write-back that occurs in this addressing mode. If it is omitted, this is the Unindexed addressing mode (see *VFP load/store multiple - Unindexed* on page C5-26) instead.

<registers>   Specifies the list of registers loaded or stored by the instruction. For details of which
                     registers are specified and how Fd, D and offset are set, see individual instructions.

Architecture version

All

Operation

```
if (offset[0] == 1) and (cp_num == 0b1011) then  /* FLDMX or FSTMX */
    word_count = IMPLEMENTATION DEFINED value (<= offset)
else                                             /* Others */
    word_count = offset
start_address = Rn
end_address = start_address + 4 * word_count - 4
if ConditionPassed(cond) then
    Rn = Rn + 4 * offset
```

Usage

For FLDMD, FLDMS, FSTMD and FSTMS, this addressing mode can be used to load or store an element of
an array of short vectors and advance the pointer to the next element. For example, if Rn points to an element
of an array of graphics points, each consisting of four single-precision co-ordinates, then:

```
FSTMIAS  Rn!, {s16-s19}
```

stores the single-precision registers s16, s17, s18 and s19 to the current element of the array and advances
Rn to point to the next element.

A related use occurs with long vectors of floating-point data. If Rn points to a long vector of single-precision
values, the same instruction stores s16, s17, s18 and s19 to the next four elements of the vector and advance
Rn to point to the next element after them.

For FSTMD, FSTMS and FSTMX, this addressing mode is useful for pushing register values on to an Empty
Ascending stack. Use FSTMD or FSTMS respectively when it is known that the registers contain only
double-precision data or only single-precision data. Use FSTMX when the precision of the data held in the
registers is unknown, and nothing needs to be done with the stored data apart from reloading it with a
matching FLDMX instruction. For instance, for callee-save registers in procedure entry sequences.

If multiple registers holding values of known but different precisions need to be pushed on to a stack, FSTMX
can be used if nothing needs to be done with the stored data apart from reloading it with a matching FLDMX
instruction. Otherwise, a sequence of FSTMD and FSTMS instructions needs to be used.

For FLDMD, FLDMS and FLDMX, this addressing mode is useful for popping data from a Full Descending
stack. The choice of which instruction to use follows the same principles as above.

Notes

Offset restrictions   The offset value must at least 1 and at most 33. If the offset is 0 or greater than 33, the instruction is always UNPREDICTABLE. Each instruction also imposes further restrictions on the offset, depending on the values of Fd and D. See the individual instruction descriptions for details of these.

ARM DDI 0100F-02

C5.5.4   VFP load/store multiple - Decrement
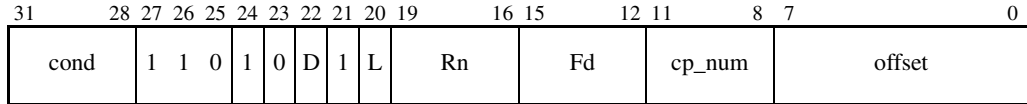
31		28 27	26	25	24	23	22	21	20	19		16 15		12 11		8 7		0
cond		1	1	0	1	0	D	1	L	Rn		Fd		cp_num		offset		

This addressing mode is for VFP load multiple and store multiple instructions, and forms a range of addresses. The first address formed is the start_address, and is the value of the base register Rn minus four times the offset. Subsequent addresses are formed by incrementing the previous address by four.

- For the FLDMS and FSTMS instructions, the offset in the instruction is equal to the number of single-precision registers to be transferred. One address is generated for each register, so the end_address is four less than the value of the base register Rn.

- For the FLDMD and FSTMD instructions, the offset in the instruction is equal to twice the number of double-precision registers to be transferred. Two addresses are generated for each register, so the end_address is four less than the value of the base register Rn.

- For the FLDMX and FSTMX instructions, the offset in the instruction is one more than twice the number of double-precision registers to be transferred.

The number of addresses generated is at most equal to the offset, but can be a smaller number (decided by the implementor) provided the FLDMX and FSTMX instructions function correctly (see *FLDMX* on page C4-43 and *FSTMX* on page C4-108). Accordingly, the end_address is the value of the base register Rn, minus an IMPLEMENTATION DEFINED amount which is at least four.

For all instructions, if the condition specified in the instruction matches the condition code status, Rn is decremented by four times the offset specified in the instruction. The conditions are defined in *The condition field* on page A3-7.

Instruction syntax

```
<opcode>DB<precision>{<cond>}  <Rn>!, <registers>
```

where:

<opcode>   Is FLDM or FSTM, and controls the value of the L bit.

<precision>

             Is D, S or X, and controls the values of cp# and offset[0].

<cond>   Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-7. If <cond> is omitted, the AL (always) condition is used.

<Rn>   Specifies the base register. If R15 is specified for <Rn>, the instruction is UNPREDICTABLE.

!   indicates the base register write-back that occurs in this addressing mode. It cannot be omitted, as no non-write-back variant of this addressing mode exists.

<registers>

             Specifies the list of registers loaded or stored by the instruction. See the individual instructions for details of which registers are specified and how Fd, D and offset are set in the instruction.

Copyright © 1996–2003 ARM Limited. All rights reserved.
Confidential - Draft

Architecture version

All

Operation

```
if (offset[0] == 1) and (cp_num == 0b1011) then  /* FLDMX or FSTMX */
    word_count = IMPLEMENTATION DEFINED value (<= offset)
else                                             /* Others */
    word_count = offset
start_address = Rn - 4 * offset
end_address = start_address - 4 * offset + 4 * word_count - 4
if ConditionPassed(cond) then
    Rn = Rn - 4 * offset
```

Usage

For FSTMD, FSTMS and FSTMX, this addressing mode is useful for pushing register values on to a Full Descending stack. Use FSTMD or FSTMS respectively when it is known that the registers contain only double-precision data or only single-precision data. Use FSTMX when the precision of the data held in the registers is unknown, and nothing needs to be done with the stored data apart from reloading it with a matching FLDMX instruction. For instance, for callee-save registers in procedure entry sequences.

If multiple registers holding values of known but different precisions need to be pushed on to a stack, FSTMX can be used if nothing needs to be done with the stored data apart from reloading it with a matching FLDMX instruction. Otherwise, a sequence of FSTMD and FSTMS instructions needs to be used.

For FLDMD, FLDMS and FLDMX, this addressing mode is useful for popping data from an Empty Ascending stack. The choice of which instruction to use follows the same principles as above.

For FLDMD, FLDMS, FSTMD and FSTMS, this addressing mode can also be used in code that scans backwards through long vectors or through arrays of short vectors. In each case, it causes a pointer to an element to be moved backwards past a set of values and loads that set of values into registers.

Notes

Offset restrictions   The offset value must at least 1 and at most 33. If the offset is 0 or greater than 33, the instruction is always UNPREDICTABLE. Each instruction also imposes further restrictions on the offset, depending on the values of Fd and D. See the individual instruction descriptions for details of these.

C5.5.5 VFP load/store multiple addressing modes (alternative names)

Like the ARM load and store multiple addressing modes, these addressing modes are useful for accessing stacks, but the load (pop) and store (push) instructions need to use different addressing modes. See *Load and Store Multiple addressing modes (alternative names)* on page A5-58 for more details.

As for the ARM instructions, alternative addressing mode names are provided which are more applicable to stack operations. FD and EA are used respectively to denote instructions suitable for Full Descending stacks and Empty Ascending stacks.

Table C5-2 shows the relationship between the non-stacking and stacking names of the instructions:

Table C5-2 VFP load/store multiple addressing modes

Non-stacking mnemonic	Stacking mnemonic
FLDMIAD	FLDMFDD
FLDMIAS	FLDMFDS
FLDMIAX	FLDMFDX
FLDMDBD	FLDMEAD
FLDMDBS	FLDMEAS
FLDMDBX	FLDMEAX
FSTMIAD	FSTMEAD
FSTMIAS	FSTMEAS
FSTMIAX	FSTMEAX
FSTMDBD	FSTMFDD
FSTMDBS	FSTMFDS
FSTMDBX	FSTMFDX

—— **Note** ——

No mnemonics are provided for Full Ascending or Empty Descending stack types, because the VFP load multiple and store multiple addressing modes do not support these types efficiently. This is a consequence of the fact that the LDC and STC addressing modes do not support these modes efficiently (see *Addressing Mode 5 - Load and Store Coprocessor* on page A5-60).

It is therefore recommended that these stack types are not used on systems that use the VFP architecture.

Part D
Debug Architecture

Chapter D1
Introduction to the Debug Architecture

This chapter gives an introduction to the debug architecture. It contains the following sections:

- *Introduction* on page D1-2
- *Trace* on page D1-8
- *Debug and ARMv6* on page D1-10.

D1.1 Introduction

ARMv6 is the first version of the architecture to include debug provisions. Prior to this debug was an accessory. Some defacto standards have emerged, but these have not previously been formally recognized. Debug developed from simple monitor based solutions, through *In Circuit Emulation* (ICE) based support, and most recently program and data flow trace provisions. This section introduces the *Debug Programmers' Model* (DPM) for ARM processors, including how it has evolved in ARM implementations since ARMv4. The basic models, the facilities, and changes between architectural and core versions is covered. The last sub-section provides some information on the direction that DPM is going for ARM based systems.

There are three ways to debug an ARMv6 compliant core:
- JTAG based Halted debug-mode debugging (clock stopped)
- JTAG based Monitor debug-mode debugging
- ROM Monitor using application communication channels.

——— **Note** ———

StrongARM® implementations have no JTAG support. Debug is monitor-based over channels in IO space, for example a UART or Ethernet link. The first Xscale® implementation is a hybrid monitor/ICE solution with a JTAG communications channel, debug hardware in coprocessor space, and system monitor software that is typically locked into on-chip memory.

D1.1.1 JTAG-based Halted debug-mode

JTAG-based Halted debug-mode was introduced at ARMv4 with the ARM7 family. The *D* and *I* letters in the ARM7TDMI relate to Debug and ICE. The JTAG based Halted debug-mode approach uses JTAG to access and control the processor, as well as to provide extended functionality through the facilities of the Embedded ICE module. In addition, Halted debug-mode means that the processor clock is stopped except under the control of debug.

Halted debug-mode debug is divided into 5 services:

- Control and access to critical signals affecting the core's state. In particular are control signals to force the processor into debug state on the next instruction as well as control signals that affect how the outside system perceives the state of the core. Access is provided to indicate what state the processor is in (that is, halted or running, and ARM or Thumb).

- Ability to insert instructions into the pipeline to execute a limited set of instructions. The instruction insertion approach is used to read and write registers and memory as well as to allow exit from halted state.

- Ability to read or write a low speed version of the data value bus. This mechanism combined with instruction insertion allows reading and writing registers, and by extension memory.

- Hardware comparators used for breakpoints and watchpoints and trapping of exception vectors. Depending on the core architecture version and the embedded ICE version, the exact mechanism varies. More details are provided later in this section.

- *Debug Communications Control* (DCC). A bi-directional, full duplex 32-bit word-wise channel is provided to allow communication with a running application. In Halted debug-mode (see JTAG Monitor debug-mode for other uses), DCC is normally used for logging from application to host, and for semi-hosting services. The specifics of DCC are covered in more detail later in this section.

Halted debug-mode debugging is the most widely used form of run/stop debugging in ARM implementations. Its most significant benefits are:

- It uses the informal standard form of IEEE JTAG. Although it is not fully JTAG compliant, it is sufficiently compliant to match standards adopted for debugging.

- It does not use application-focused communication channels such as RS-232, Ethernet, or USB.

- It does not require any memory on the target to operate. Some debuggers may use some target memory to allow access to certain facilities (for example coprocessors or flash programming), but memory allocation is not a requirement for use.

- It has hardware comparators that allow stopping execution under controlled and possibly complex conditions.

- With a cross-triggering module, it allows minimal skew stopping of multiple cores at the same time.

- Once stopped, it cannot be affected by outside influences such as interrupts.

- It controls when it is accessing memory, so side effects can be predicted and controlled.

- In some core versions it can protect the L1 cache state to allow timing measurements.

The most significant issues with Halted debug-mode debugging are:

- The external system usually does not handle transition to this state and so it can fail, often catastrophically, when the core is halted. Issues include:
    — overrun of peripherals
    — watchdogs allowed to timeout (unless constantly reset by the debugger)
    — failed or lost transactions.

- Interrupts are prevented and no special code can execute, so applications that need an orderly shutdown or stop cannot be accommodated.

- Any stop is very time-intrusive, and so Halted debug-mode cannot be used to poll/sample the processor or make conditional stops without significant impact on application behavior.

D1.1.2   JTAG-based Monitor debug-mode

JTAG-based Monitor debug-mode was introduced as an upgrade to some members of the ARM7 and ARM9 family. The JTAG-based Monitor debug-mode approach uses JTAG as a communication device to a monitor, as well as providing access to extended functionality through the facilities of the *Embedded ICE* (EICE) module. Only processors with the EICE-RT (RT stands for realtime) module may use the full services of EICE with Monitor debug-mode. That is, the hardware comparators are only available if the EICE module used supports Monitor debug-mode. With older versions of EICE, Monitor debug-mode is still supported, but without the full capabilities.

It is notable that it is possible to mix Halted debug-mode debug with Monitor debug-mode. The most notable example is to allow OS (or RTOS) Running System Debug (RSD) using Monitor debug-mode, but with Halted debug-mode debugging available as a fallback for system failure and boot time debug. The ability to switch between these two debug-modes is fully supported by the processor.

Monitor debug-mode debug is divided into the following services:

* Access to the Halted debug-mode control signals are still provided to allow reverting to Halted debug-mode debug, such as on total system failure.

* Hardware comparators are used for breakpoints and watchpoints and trapping of exception vectors. Depending on the core architecture version and the embedded ICE version, the exact mechanism varies. More details are provided later in this section. Unlike Halted debug-mode, these mechanisms are used to enter a monitor on the target (via exceptions) instead of stopping the processor.

* DCC. A bi-directional, full duplex 32-bit word-wise channel is provided to allow communication with a monitor running on the target. The exact protocol is defined between the debugger and the monitor. Depending on system design, the host sending a word to the target might or might not be permitted to interrupt the processor and enter the monitor.

Monitor debug-mode debugging using JTAG is only heavily used in certain application segments. Its most significant benefits are:

* It allows interrupts to be serviced by the application while stopped in debug. This is the most compelling reason for using JTAG-based Monitor debug-mode for certain applications, where not servicing the interrupt could cause serious physical harm to the system.

* The monitor can have code on stop entry (break or otherwise) which gracefully shuts down or freezes the external system.

* It can be used to poll the status of the target with minimal intrusion.

* It uses an informal standard form of IEEE JTAG. Although it is not fully JTAG compliant, it is sufficiently compliant to match standards adopted for debugging.

* It does not use application-focused communication channels such as RS-232, Ethernet, or USB.

* With newer processor cores, availability of hardware comparators that allow stopping execution under controlled and possibly complex conditions.

The most significant issues with Monitor debug-mode debugging are:

• The monitor executes in the same memory as the application. This means both that it is using application memory (so extra must be available) and that it can be overwritten/corrupted by an errant application.

• Not all systems have DCC tied to interrupts, so they cannot gain control of the monitor unless it is asked to poll by the application. For applications that use idle loops, idle tasks, or rotational event models, it is possible to have the application call the monitor to do this polling, but this means that the monitor is part of the application. When interrupts are tied in this way, you must mix application interrupt servicing with debug interrupt servicing, unless a *Vectored Interrupt Controller* (VIC) is used.

• Performance can be impacted significantly by the ability of the host to poll the DCC channel (from JTAG) as well as handshakes needed in the protocol.

D1.1.3 ROM Monitor

The traditional debugging model for the ARM processor was use of a ROM monitor. A ROM monitor is code that runs on the unprotected processor, catches some or all exceptions, and communicates using an application-style communication device. Common communication devices include RS-232 (serial), Ethernet (often shared with the application), and proprietary communication schemes used in the application space (for example CAN).

Until recent versions of the ARM cores, ROM monitors received no special support from the processor or EICE module. That is, ROM monitors did not have access to comparators used for breakpoints and watchpoints, single step support (when available at all), or ETM trace (see *Trace* on page D1-8). In newer versions, some or all of these facilities are exposed to the ROM monitor.

Typically ROM monitors are used for three reasons:

- Part of boot code for an OS. The monitor might also support terminal interaction with a user.

- No JTAG support in the chip, or JTAG is not feasible for the situation. In some chips, JTAG is not pinned out at all. In other cases, JTAG is reasonable for early development, but not for in-field use, for example because of board constraints.

- Part of an OS proper. Some operating systems (for example Linux) and some subsystems (for example Java VM) have a monitor agent built in. In many of these cases, the monitor could be changed to use DCC and JTAG (see *JTAG-based Halted debug-mode* on page D1-2) instead of relying on OS services such as TCP/IP.

Support for ROM monitors falls into 4 services:

- A mechanism to support patched breakpoints in ARM and Thumb state. In general, this is supported using an undefined instruction, which is defined to be unique to this use. Some monitors can instead choose to use SWI (software interrupt).

- The ability of a supervisor mode application (monitor) to see the values of other modes. Although this facility is naturally part of the ARM architecture, it is critical for a monitor to represent the state of an application under debug.

- In newer cores, the ability to enable the EICE module comparators to cause abort exceptions on matching of specific criteria. With the ability to set these via the CP14 coprocessor interface, JTAG is not required, so a ROM monitor can set them directly.

- In newer cores, the ability to control the ETM registers. ETM trace must be set-up using an interface other than its trace port, which is output only. In older cores, this was done via JTAG. In newer cores, a ROM monitor can access these registers to program the trace for capture by an external device or internal buffer.

ROM Monitor debug-mode's significant benefits are:

- It allows interrupts to be serviced by the application while the application is stopped in debug.

- The monitor can have code on stop entry (break or otherwise) which gracefully shuts down or freezes the external system.

- It can be part of the OS so that it allows controlling specific processes or threads only, while allowing others to continue running.

- If it is an OS monitor, it can share the communication channel used by the application (such as TCP/IP).

- It can be used to poll the status of the target with minimal intrusion.

- With newer processor cores, hardware comparators and ETM registers are available that can be set up from the monitor.

The most significant issues with Monitor debug-mode debugging are:

- The monitor executes in the same memory as the application. This means both that it is using application memory (so extra must be available) and that it can be overwritten/corrupted by an errant application.

- It uses application-focused communication channels such as RS-232, Ethernet, or USB. This not only contends for a needed resource in many cases, but also contends for needed interrupts. Without interrupt-driven support from the channel peripheral, the monitor cannot be used to stop the application from the host.

- With older cores, there is no access to comparators or ETM.

- If interrupts are disabled by an errant application, the monitor cannot gain control (unless it hits a breakpoint, exception, or hardware watchdog).

D1.2 Trace

All current architecture implementations from ARM support trace using *Embedded Trace Macrocell* (ETM). The fundamental requirements of ETM are:

* to compress the output to require fewer pins or the smallest buffer possible
* to support a high degree of control over what gets traced and when
* to allow ARM silicon developers to choose how much functionality they want, considering the gate count trade off
* to support different pin signalling methods to accommodate different chip requirements.

The ETM architecture is available from ARM to licensees and third party tools vendors wishing to support its standard interfaces.

——— **Note** ———

The Trace Architecture and details of the ETM are documented separately from this manual.

ETM is currently at version 3. Version 3 is very similar to earlier versions in the trace triggering and control area, but very different in the output format. The output format is not normally a concern to anyone other than a tool vendor writing a decompressor. However, there are some notable differences in the ETM version 3 output format design:

* It is designed to compress instructions much better.

* It uses headers on each trace to allow for new types of packets. The headers also allows a capture device to know when the end of the data for that packet occurs. This in turn allows serializing the packets using fewer pins.

* It accommodates other sources of trace when coupled with a *funnel*. A funnel allows other sources of trace, including for example:

  — other ARM cores

  — DSP cores

  — AMBA trace devices

  — ASICs.

  The funnel combines the trace sources using the packets to determine where each stream ends.

* It can have data only (no instructions).

* It can tag architecture version 6 details such as endianness, unaligned, and ASID changes.

There are two main forms of trace capture:
* a *Trace Port Analyzer* (TPA), see *Trace Port Analyzer* on page D1-9
* a Trace Buffer, see *On-chip Trace Buffer* on page D1-9.

D1.2.1 Trace Port Analyzer

A TPA is an external piece of hardware that captures ETM signals coming off the chip. The TPA must be configured to support the various methods of pin handling. The TPA can also choose trigger capture policies and whether to add time stamps to the output.

The ETM pin-signalling model supports a variety of methods that trade-off pin clocking speeds against number of pins. The chip vendor chooses a pin signalling method that makes the most sense for the chip, the board environment, and the application space. The highest core clock and ETM capture rates require the use of an on-chip buffer. However, new interface specifications are constantly being developed to support increasing speeds of external interface reflecting advances in process technology. These newer specifications will cover methodologies for the capture devices as well as the chips.

D1.2.2 On-chip Trace Buffer

An on-chip Trace Buffer is a block of memory that acts as a ring buffer to hold the trace data. It can be read after tracing has stopped, or it can be off-loaded while still tracing. The buffer design can also be used as a large FIFO and connected to a TPA port or funnel to allow high burst rates but slower sustained rates.

The trace buffer model treats the buffer as an AMBA slave so that it can be drained by a variety of devices. This can allow it to be used as system memory when not performing the trace function.

D1.3 Debug and ARMv6

Debug in ARMv6 is primarily concerned with defining a coprocessor interface (using coprocessor 14) to allow a level of debug software access and control of the system typically only available over a JTAG based solution in earlier implementations. The following sections in this part define the ARMv6 features and usage model.

D1.3.1 Debug and virtual addresses

Unless otherwise stated, all the addresses referred to in the following chapters are Virtual Addresses (VAs) as described in B8.TBD.

The terms *Instruction Virtual Address* (IVA) and *Data Virtual Address* (DVA) are used to mean the VA corresponding to the instruction address and data address respectively.

Chapter D2
Coprocessor 14, the Debug Coprocessor

This chapter gives information about Coprocessor 14, the Debug Coprocessor. It contains the following sections:

- *Coprocessor 14 debug registers* on page D2-2
- *Coprocessor 14 debug instructions* on page D2-5
- *Debug register reference* on page D2-8

D2.1 Coprocessor 14 debug registers

Table D2-1 shows the set of CP14 debug registers.

To access the CP14 debug registers, opcode_1 and CRn must be set to 0. The opcode_2 and CRm fields of the coprocessor instructions are used to encode the CP14 debug register number, where the register number is {opcode2, CRm}.

Table D2-1 CP14 debug register map

Binary address		Register number	Abbreviation [a]	CP14 debug register name
Opcode_2	CRm			
000	0000	0	DIDR	Debug ID Register
000	0001	1	DSCR	Debug Status and Control Register
000	0010-0100	2-4	-	RESERVED
000	0101	5	DTR	Data Transfer Register
000	0110	6	-	RESERVED
000	0111	7	VTR	Vector Trap Register
000	1000-1111	8-15	-	RESERVED
001-011	0000-1111	16-63	-	RESERVED
100	0000-1111	64-79	BVRy	Breakpoint Value Registers / RESERVED
101	0000-1111	80-95	BCRy	Breakpoint Control Registers / RESERVED
110	0000-1111	96-111	WVRy	Watchpoint Value Registers / RESERVED
111	0000-1111	112-127	WCRy	Watchpoint Control Registers / RESERVED

a. y is the decimal representation of the binary number CRm.

To set a Breakpoint Debug Event, two registers are needed: a BVR and a BCR. BCRy is the corresponding control register for BVRy. A pair of breakpoint registers BVRy/BCRy is called a *Breakpoint Register Pair* (BRP). The BVR of a BRP is loaded with an IVA and then its contents can be compared against the IVA bus of the processor.

Similarly, a pair of watchpoint registers WVRy/WCRy is called a *Watchpoint Register Pair* (WRP). The WVR of a WRP is loaded with a DVA (see *Debug and virtual addresses* on page D1-10) and then its contents can be compared against the DVA bus of the processor.

Space is reserved in the address map for up to 16 BRPs and 16 WRPs.

ARMv6 compliant cores have support for thread-aware breakpoints and watchpoints. A context ID can be loaded into the BVR and the BCR can be configured so this BVR value is compared against the CP15 Context ID (register 13) instead of the IVA bus. Another register pair loaded with an IVA or DVA can then be linked with the context ID-holding BRP. A Breakpoint/Watchpoint Debug Event is only generated if both the address and the Context ID match at the same time. In this way, unnecessary hits can be avoided when debugging a specific thread within a task.

Breakpoint Debug Events generated on context ID matches only are also supported. However, if the match occurs while the processor is running in a privileged mode and the Debug logic in Monitor Debug-Mode, it is ignored. This avoids the processor ending in an unrecoverable state.

It is not mandatory that all the BRPs have context ID comparison capability. A particular ARMv6 compliant core can implement:

1.   Any number of BRPs from 2 to 16. Where DIDR[27:24] = n and n>=1, the number of BRPs supported equals (n+1). This is the total number of BRPs, including context ID capable and non-capable.

2.   Any number of WRPs from 1 to 16. Where DIDR[31:28] = n, the number of WRPs supported equals (n+1).

3.   Any number of BRPs with context ID comparison capability from 1 to the implemented number of BRPs. Where DIDR[23:20] = n, the number of context ID capable BRPs equals (n+1).

Registers that are not implemented are Reserved, that is, they cannot be used for any other purpose.

The implemented register pairs must take numbers as follows:

•   The implemented BRPs start at 0. For example, BRP0 to BRP5 in the case that 6 BRPs are implemented (DIDR[27:24]=0b0101).

•   The implemented WRPs also start at 0. For example, WRP0 to WRP1 in the case that 2 WRPs are implemented (DIDR[31:28]=0b0001).

•   The BRPs with context ID comparison capability occupy the higher BRP numbers. For example, BRP4 to BRP5 in the case that 6 BRPs are implemented and 2 of them have context ID comparison capability (DIDR[27:24]=0b0101 and DIDR[23:20]=0b0001).

D2.1.1   The minimum number of BRPs and WRPs

Any implementation must have a minimum of 2 BRPs and 1 WRP. At least 1 BRP must have context ID comparison capability.

This is to guarantee that at least the following can be done:
•   set an unlinked breakpoint on IVA
•   set an unlinked breakpoint on a context ID value
•   set a linked breakpoint
•   set an unlinked watchpoint
•   set a linked watchpoint.

However, ARM recommends that at least 3 BRPs and 1 WRP are implemented; 1 BRP must have context ID comparison capability. The additional BPR can be dedicated to single-stepping (that is, to point to the following instruction to the one where the application is stopped) while the rest of the resources are free to program any of these Debug Events.

D2.2 Coprocessor 14 debug instructions

This section describes all the CP14 debug instructions that must be implemented.

The legal CP14 debug instructions are shown in Table D2-2.

Table D2-2 Legal CP14 debug instructions

Binary address		Register number	Abbreviation	Legal instructions [a]
Opcode_2	CRm			
000	0000	0	DIDR	`MRC p14,0,Rd,c0,c0,0`

000	0001	1	DSCR	`MRC p14,0,Rd,c0,c1,0`
				`MRC p14,0,R15,c0,c1,0`
				`MCR p14,0,Rd,c0,c1,0`

000	0101	5	DTR	`MRC p14,0,Rd,c0,c5,0`
				`MCR p14,0,Rd,c0,c5,0`
				`STC p14,c5,<addressing_mode>`
				`LDC p14,c5,<addressing_mode>`

000	0111	7	VTR	`MRC p14,0,Rd,c0,c7,0` `MCR p14,0,Rd,c0,c7,0`
100	0000-1111	64-79	BVR	`MRC p14,0,Rd,c0,cy,4` `MCR p14,0,Rd,c0,cy,4`
101	0000-1111	80-95	BCR	`MRC p14,0,Rd,c0,cy,5` `MCR p14,0,Rd,c0,cy,5`
110	0000-1111	96-111	WVR	`MRC p14,0,Rd,c0,cy,6` `MCR p14,0,Rd,c0,cy,6`
111	0000-1111	112-127	WCR	`MRC p14,0,Rd,c0,cy,7` `MCR p14,0,Rd,c0,cy,7`

a. Rd is any general-purpose ARM register, R0-R14. y is the decimal representation of the binary number CRm.

In Table D2-2, `MRC p14,0,Rd,c0,c5,0` and `STC p14,c5,<addressing mode>` refer to the rDTR, and `MCR p14,0,Rd,c0,c5,0` and `LDC p14,c5,<addressing mode>` refer to the wDTR. See *Register 5, Data Transfer Register (DTR)* on page D2-15 for more details.

D2.2.1 Transferring DSCR flags to CPSR flags

MRC p14,0,R15,c0,c1,0 instruction sets the CPSR flags as follows:

- N flag = DSCR[31]. This is an UNPREDICTABLE value.
- Z flag = DSCR[30]. This is the value of the rDTRfull flag.

- C flag = DSCR[29]. This is the value of the wDTRfull flag.
- V flag = DSCR[28]. This is an UNPREDICTABLE value.

You can use the CPSR flags to control following conditional instructions in the usual way.

D2.2.2   Executing CP14 debug instructions

Table D2-3 shows the results of executing CP14 debug instructions.

Table D2-3 Results of CP14 debug instruction execution

Processor mode	Debug state	DSCR[15:14] (Mode enabled and selected)	DSCR[12] (DCC User accesses disable)	Read DIDR, read DSCR, read/write DTR	Write DSCR	Read/write other registers
x	Yes	xx	x	Proceed	Proceed	Proceed
User	No	xx	0	Proceed	UNDEFINED exception	UNDEFINED exception
User	No	xx	1	UNDEFINED exception	UNDEFINED exception	UNDEFINED exception
Privileged	No	00 (none)	x	Proceed	Proceed	UNDEFINED exception
Privileged	No	01 (Halt)	x	Proceed	Proceed	UNDEFINED exception
Privileged	No	02 (Monitor)	x	Proceed	Proceed	Proceed
Privileged	No	03 (Halt)	x	Proceed	Proceed	UNDEFINED exception

Not implemented

If the processor tries to execute a CP14 debug instruction that either is not in Table D2-2 on page D2-5, or is targeted to a RESERVED register such as a non-implemented BVR, the Undefined Instruction exception is taken.

Debug logic reset

If the processor tries to execute a CP14 debug instruction while the debug logic reset is activated, this instruction proceeds normally. However:

- CP14 debug instructions that read CP14 debug registers returns the reset value of those registers. See *Reset values of the CP14 debug registers* on page D2-25. Reset values of the CP14 debug registers on page 20 for details.

- CP14 debug instructions that write to CP14 debug registers do not have any effect. They execute but, because debug logic reset is activated, the CP14 debug registers keep their reset values.

Privilege

Access to the Comms Channel (read DIDR, read DSCR and read/write DTR) must be possible in User mode. All other CP14 debug instructions are privileged. If the processor tries to execute one of these in User mode, the Undefined Instruction exception is taken.

If the DSCR[12] User mode access to Comms Channel disable bit is set, all CP14 debug instructions are considered as privileged and any User mode access to any CP14 debug register generates an Undefined Instruction exception.

Debug state

If the core is in Debug state (see *Debug state* on page D3-8), any CP14 debug instruction can be executed regardless of the processor mode.

Value of the DSCR[15:14] bits

Once the DSCR[14] bit is set (Halted debug-mode selected and enabled), if the software running on the processor tries to access any other register different from the DIDR, the DSCR or the DTR, the core takes the Undefined Instruction exception. The same thing happens if the core is not in any Debug Mode (DSCR[15:14]=0b00).

This lockout mechanism ensures that the software running on the core cannot modify the settings of a Debug Event programmed by the External Debugger.

D2.2.3 CP14 debug instructions and Instruction Memory Barriers

——— **Note** ———

Writes to CP14 debug registers can take effect a number of instructions after the corresponding MCR/LDC. An Instruction Memory Barrier (IMB) (see *Memory Barriers* on page B2-29) should be inserted before the change of state can be relied on.

———

D2.3 Debug register reference

The following codes and terms are used in this section:

R   Read-only. Written values are ignored. Must be written as 0, or preserved by writing the same value previously read from the same fields on the same processor.

W   Write-only. A read to this bit returns an UNPREDICTABLE value.

RW   Read/Write.

C   Clear on read. Cleared every time the register is read.

UNP/SBZP   UNPREDICTABLE/Should-Be-Zero-or-Preserved. A read to this bit returns an UNPREDICTABLE value. Must be written as 0, or preserved by writing the same value previously read from the same fields on the same processor. These bits are usually RESERVED for future

expansion.

RAZ   Read As Zero. A read to this bit returns 0.

Core view   This column defines the core access permission for a given bit.

External view This column defines what the External Debugger view of a given bit must be.

Read/write attributes

This is used when the core and the External Debugger view are the same.

-   UNPREDICTABLE.

D2.3.1   Register 0, Debug ID Register (DIDR)

Table D2-4 shows the layout of the Debug ID Register.

Table D2-4 Debug ID Register bit definitions

Bits	Core view	External view	Value	Description [a]
[3:0]	R	R		Implementation defined revision number. This number is incremented on bug fixes.
[7:4]	R	R		Implementation defined variant number. This number is incremented on functional changes.
[15:8]	UNP/SBZP	UNP/SBZP		RESERVED
[19:16]	R	R	0x1	Debug Architecture Version: This Debug Architecture version
[23:20]	R	R	0000 0001 ... 1111	Implemented BRPs with context ID comparison capability: 1 BRP has context ID comparison capability 2 BRPs have context ID comparison capability ... 16 BRPs have context ID comparison capability
[27:24]	R	R	0000 0001 0010 ... 1111	Number of implemented BRPs: RESERVED (the minimum number of BRPs is 2) 2 implemented BRPs 3 implemented BRPs ... 16 implemented BRPs
[31:28]	R	R	0000 0001 ... 1111	Number of implemented WRPs: 1 implemented WRP 2 implemented WRPs ... 16 implemented WRPs

a.   BRP: Breakpoint Register Pair, WRP: Watchpoint Register Pair

The values of the fields of this register are IMPLEMENTATION DEPENDENT. However, the chosen values must agree with the ones in the CP15 Main ID Register:

*   DIDR[3:0] must equal the Main ID Register bits[3:0]
*   DIDR[7:4] must equal the Main ID Register bits[23:20].

See *Register 0: ID codes* on page B3-7 for a description of the Main ID Register.

D2.3.2 Register 1, Debug Status and Control Register (DSCR)

Table D2-5 shows the layout of the Debug Status and Control Register.

Table D2-5 Debug ID Register bit definitions

Bits	Core view	External view	Reset value	Description [a]
0	R	R	[b]	Core halted.
1	R	R	[a]	Core restarted.
[5:2]	RW	R	-	Method of debug entry.
6	R	RC	0	Sticky precise abort bit.
7	R	RC	0	Sticky imprecise abort bit.
[9:8]	UNP/SBZP	UNP/SBZP	-	RESERVED
10	R	RW	0	DbgAck: debug acknowledge.
11	R	RW	0	Interrupts disable.
12	RW	R	0	User mode access to Comms channel disable.
13	R	RW	0	Execute ARM instruction enable.
14	R	RW	0	Halt/Monitor debug-mode select.
15	RW	R	0	Monitor debug-mode enable.
[28:16]	UNP/SBZP	UNP/SBZP	-	RESERVED
29	R	R	0[a]	wDTRfull: wDTR register full.
30	R	R	0[a]	rDTRfull: rDTR register full.
31	UNP/SBZP	UNP/SBZP	-	RESERVED

a. For further information on each, see subsections below.
b. See tbd xref

Core halted, bit[0]

The External Debugger can poll this bit to determine when the processor has entered Debug state. See *Debug state* on page D3-8 for a definition of Debug state.

0   The processor is in normal state.
1   The processor is in Debug state.

Core Restarted, bit[1]

The External Debugger can poll this bit to determine when the processor has exited Debug state. See *Debug state* on page D3-8 for a definition of Debug state.

0   The processor is exiting Debug state. This bit is read in conjunction with bit[0].

1   The processor has exited Debug state.

Method of Debug Entry, bits[5:2]

Table D2-6 shows the meanings of the method of debug entry values:

Table D2-6 Meaning of method of debug entry values

Value	Description
0000	A Debug State Entry Request command occurred
0001	Breakpoint occurred
0010	Watchpoint occurred
0011	BKPT instruction occurred
0100	External Debug Request signal activation occurred
0101	Vector catch occurred
0110	D-side abort occurred
0111	I-side abort occurred
1xxx	RESERVED

These bits are set to indicate:

• the cause of jumping to the Prefetch or Data Abort vector

• the cause of entering Debug state.

This way, a Prefetch Abort or a Data Abort handler can find out whether they should jump to the Monitor Target or not. Also, an External Debugger/Monitor Target can find out which was the specific Debug Event that caused the Debug state/Debug exception entry.

Two particular cases are:

• A BKPT instruction executed in normal state and while debug is disabled sets this field to *BKPT instruction occurred*.

• If an unaligned access causes a Data Abort exception, it sets this field to *D-side abort occurred*.

Sticky Precise Abort, bit[6]

This flag is used to detect precise data aborts generated by instructions issued to the processor using the External Debug Interface (see *Access to CP14 debug registers from the external debug interface* on page D2-26). If the DSCR[13] Execute ARM Instruction enable bit is clear the value of the Sticky Precise Abort flag is UNPREDICTABLE. This flag is cleared on reads to the DSCR by the External Debugger.

0   No precise data abort occurred since the last time this bit was cleared.

1   A precise data abort has occurred since the last time this bit was cleared.

Sticky Imprecise Abort, bit[7]

This flag is used to detect imprecise aborts generated by instructions issued to the processor using the External Debug Interface (see *Access to CP14 debug registers from the external debug interface* on page D2-26). If the DSCR[13] Execute ARM Instruction enable bit is clear the value of the Sticky Abort flag is UNPREDICTABLE. This flag is cleared on reads to the DSCR by the External Debugger.

0   No imprecise data abort occurred since the last time this bit was cleared.

1   An imprecise data abort has occurred since the last time this bit was cleared.

DbgAck, bit[10]

If this bit is set, the **DBGACK** output signal (see *External debug* on page D3-4) is forced high regardless of the processor state.

This is intended to be used by an External Debugger to execute pieces of code in normal state as part of the debugging process.

Interrupts Disable, bit[11]

If this bit is asserted, the IRQ and FIQ input signals are inhibited.

0   Interrupts enabled.

1   Interrupts disabled.

This is intended to be used by an External Debugger to execute pieces of code in normal state as part of the debugging process.

For example, to execute an OS service routine to bring a page from disk into memory and then go back to the application to see the effect that this change of state produces. It is undesirable that any interrupt gets serviced during the routine execution.

User mode access to Comms Channel disable, bit[12]

If this bit is set and a User mode process tries to access the DIDR, DSCR, or the DTR, the Undefined Instruction exception is taken.

Setting this bit means that a User mode process cannot access any CP14 debug register.

0	User mode access to Comms Channel enabled.
1	User mode access to Comms Channel disabled.

Execute ARM Instruction enable, bit [13]

0	Disabled.
1	The mechanism for forcing the core to execute ARM instructions in Debug state via the External Debug Interface is enabled.

Setting this bit when the core is not in Debug state leads to UNPREDICTABLE behavior.

——— **Note** ———

If this bit is set, the wDTRfull and rDTRfull flags must not be used for handshaking. Use bit[32] of scan chain 4 or scan chain 5 (the Instruction Completed bit) instead.

Halt/Monitor Debug-Mode Select, bit[14]

0	Monitor Debug-Mode selected.
1	Halted debug-mode selected and enabled.

Monitor Debug-Mode Enable, bit[15]

0	Monitor Debug-Mode disabled.
1	Monitor Debug-Mode enabled.

——— **Note** ———

• Monitor Debug-Mode has to be both selected and enabled (bit 14 clear and bit 15 set) for the core to take a Debug exception.

• If the external interface input **DBGEN** is low, the DSCR[15:14] read as 0b00. Programming of the bits is masked until **DBGEN** is taken high, at which time the value and behavior revert to the programmed value.

wDTRfull: wDTR register full, bit[29]

0	wDTR register empty.
1	wDTR register full.

This flag is automatically cleared on reads by the External Debugger of the wDTR and is set on writes by the core to the same register.

────── **Note** ──────

You must not use this flag for handshaking if the Execute ARM Instruction enable bit, bit[13], is high.

──────────────

rDTRfull: rDTR register full, bit[30]

0   rDTR register empty.

1   rDTR register full.

This flag is automatically set on writes by the External Debugger to the rDTR and is cleared on reads by the core of the same register. No writes to the rDTR are allowed if the rDTRfull flag is set.

────── **Note** ──────

You must not use this flag for handshaking if the Execute ARM Instruction enable bit, bit[13], is high.

──────────────

D2.3.3 Register 5, Data Transfer Register (DTR)

This register consists of two separate physical registers: the *read-only DTR* (rDTR) and the *write-only DTR (wDTR)*. Note that read and write refer to the core view. The rDTR is accessed with a MRC or STC instruction and the wDTR with MCR or LDC (see Table D2-2 on page D2-5). See Table D2-7 for further details.

Table D2-7 Data Transfer Register bit definition

Bits	Core view	External view	Reset value	Description
31:0	R	W	-	read-only Data Transfer Register
31:0	W	R	-	write-only Data Transfer Register

For details on the use of these registers in relation with the rDTRfull and wDTRfull flags *Register 1, Debug Status and Control Register (DSCR)* on page D2-10.

D2.3.4   Register 7, Vector Trap Register (VTR)

Register 7 is the *Vector Trap Register* (VTR), see Table D2-8 for details. If a bit in the VTR is set, the following sequence is initiated:

1.   the corresponding vector is prefetched and the instruction is committed for execution

2.   a Debug exception or Debug state entry can be generated (this depends on the value of the DSCR[15:14] bits; see tbd xref 6.5 Behavior of the processor on Debug Events on page 23).

——— **Note** ———

Under this model, any kind of prefetch of an exception vector can trigger a vector trap, not just exception entries.

Table D2-8 Vector Trap Register bit definition

Bits	Read/write attributes	Reset value	Description	Normal address	High vector address
0	RW	0	Vector Trap Enable – Reset	0x00000000	0xFFFF0000
1	RW	0	Vector Trap Enable – Undefined Instruction	0x00000004	0xFFFF0004
2	RW	0	Vector Trap Enable – SWI	0x00000008	0xFFFF0008
3	RW	0	Vector Trap Enable – Prefetch Abort	0x0000000C	0xFFFF000C
4	RW	0	Vector Trap Enable – Data Abort	0x00000010	0xFFFF0010
5	UNP/SBZP	-	Reserved	-	-
6	RW	0	Vector Trap Enable – IRQ	most recent IRQ address	most recent IRQ address
7	RW	0	Vector Trap Enable – FIQ	most recent FIQ address	most recent FIQ address
31:8	UNP/SBZP	-	RESERVED	-	-

If VTR bit[6] is set, the debug logic traps the vector address corresponding to the most recent IRQ interrupt that occurred while debug was enabled (DSCR[15:14] != 2b00). The reason for this is that, as described in tbd xref [6] Exception Model, the hardware can be configured so the IRQ uses vector addresses different from the standard 0x00000018/0xFFFF0018. The same applies for VTR bit[7] and the FIQ interrupts.

The update of the VTR can take effect a number of instructions after the corresponding MCR. It is only guaranteed to have taken effect by the next IMB.

D2.3.5   Registers 64-79, Breakpoint Value Registers (BVR)

Each BVR is associated with a BCR register: BVR0 with BCR0, BVR1 with BCR1, ... BVR15 with BCR15. These are BVRy and BCRy as defined in Table 1 on page 9. A pair of breakpoint registers BVRy/BCRy is called a Breakpoint Register Pair (BRP).

The breakpoint value contained in this register corresponds to either an IVA or a context ID. Breakpoints can be set either on an IVA, a context ID or an IVA/context ID pair. For the third case two BRPs have to be linked (see tbd xref sections 4.3.6 and A.2.1.2 on pages 17 and 35 respectively). A Debug Event is generated when both the IVA and the context ID pair match at the same time.

───── **Note** ─────

Not every BVR can be compared against the CP15 Context ID register 13. See *Coprocessor 14 debug registers* on page D2-2.

Table D2-9 Breakpoint Value Registers bit definition

Bits	Read/write attributes	Reset value	Description
31:0	RW	0	Breakpoint Value

tbd words on ID comparison capability.

When a context ID capable BRP is set for IVA comparison, BVR[1:0] is ignored.

D2.3.6   Registers 80-95, Breakpoint Control Registers (BCR)

Table D2-10 shows the layout of the Breakpoint Control registers.

Table D2-10 Breakpoint Control Registers bit definition

Bits	Read/write attributes	Reset value	Description [a]
[0]	RW	0	Breakpoint enable. 0: disabled. 1: enabled.
[2:1]	RW	-	Supervisor Access, see *Supervisor Access, bits[2:1]*.
[4:3]	UNP/SBZP	-	RESERVED
[8:5]	RW	-	Byte address select, see *Byte address select, bits[8:5]*.
[15:9]	UNP/SBZP	-	RESERVED
[19:16]	RW	-	Linked BRP number, see *Linked BRP number, bits[19:16]* on page D2-19.
[20]	RW	-	Enable linking, see *Enable linking, bit[20]* on page D2-19.
[21]	RW (RAZ)	- (-)	Meaning of BVR, *Meaning of BVR, bit[21]* on page D2-19.
[31:22]	UNP/SBZP	-	RESERVED

a.   For further information on each, see subsections below.

Supervisor Access, bits[2:1]

You can condition the breakpoint to the privilege of the access being done:

00	Reserved
01	Privileged
10	User
11	Either

If this BRP is programmed for context ID comparison and linking (BCR[21:20] == 0b11), the BCR[2:1] field of the BRP that holds the IVA takes precedence. It is undefined whether this field is included in the comparison or not. In this case, set it to *Either*.

The WCR[2:1] field of a WRP linked with this BRP also takes precedence over this field.

Byte address select, bits[8:5]

The BVR is programmed with a word address. You can use this field to program the breakpoint so it hits only if certain byte addresses are accessed:

0000	The breakpoint never hits.
xxx1	The breakpoint hits if the byte at address BVR[31:2]+0 is accessed.

xx1x   The breakpoint hits if the byte at address BVR[31:2]+1 is accessed.

x1xx   The breakpoint hits if the byte at address BVR[31:2]+2 is accessed.

1xxx   The breakpoint hits if the byte at address BVR[31:2]+3 is accessed.

This field must be set to 0b1111 when this BRP is programmed for context ID comparison (BCR[21:20] == 0b1x). Otherwise Breakpoint or Watchpoint Debug Events might not be generated as expected.

——— **Note** ———

- These are little-endian byte addresses. This ensures that a breakpoint is triggered regardless of the endianness of the instruction fetch. For example, if a breakpoint is set on a Thumb instruction by setting BCR[8:5] = 0b0011, it is triggered if fetched little-endian with IVA[1:0] = 0b00, or if fetched big-endian with IVA[1:0] = 0b10.

- Breakpoints in Java state are taken only if the breakpointed address is fetched as an opcode. Breakpoints on operands are ignored.

Linked BRP number, bits[19:16]

The binary number encoded here indicates another BRP to link this one with. If a BRP is linked with itself, it is UNPREDICTABLE whether a Breakpoint Debug Event is generated or not.

Enable linking, bit[20]

0   Linking disabled.

1   Linking enabled.

When this bit is set this breakpoint is linked. See tbd xref Table 11 below for details.

Meaning of BVR, bit[21]

0   Instruction Virtual Address. The corresponding BVR is compared against the IVA bus.

1   Context ID. The corresponding BVR is compared against the CP15 Context ID (register 13).

If this BRP does not have context ID comparison capability, this control bit does not apply and is RAZ.

——— **Note** ———

The BCR[8:5] and BCR[2:1] fields still apply when a BRP is set for context ID comparison. See tbd xref section A.2.1 on page 35 for detailed programming sequences for linked breakpoints and linked watchpoints.

The number of BRPs that can be compared against the CP15 Context ID register is implementation defined. This is indicated by the DIDR[23:20] field (see *Coprocessor 14 debug registers* on page D2-2).

Meaning of bits[21:20]

00   Compare this BVR against the IVA bus. This BRP is not linked with any other one. Generate a Breakpoint Debug Event on an IVA match.

01   Compare this BVR against the IVA bus. This BRP is linked with the one indicated by BCR[19:16]. Generate a Breakpoint Debug Event on a joint IVA and context ID match.

10   Compare this BVR against the CP15 Context ID (register 13). This BRP is not linked with any other one. Generate a Breakpoint Debug Event on a context ID match.

11   Compare this BVR against the CP15 Context ID (register 13). Another BRP (of the BCR[21:20]=0b01 type) or WRP (with WCR[20]=0b1) is linked with this BRP. Generate a Breakpoint/Watchpoint Debug Event on a joint IVA/DVA and context ID match.

ARMv6 breakpoint debug event generation

This is how an ARMv6 Debug Architecture implementation must behave in generating Breakpoint Debug Events:

- The update of a BVR or a BCR can take effect a number of instructions after the corresponding MCR. It is only guaranteed to have taken effect by the next IMB.

- Updates of the CP15 Context ID register 13 can take effect a number of instructions after the corresponding MCR. However, it must be guaranteed that the write has occurred before the end of the exception return. This is to ensure that a User mode process, switched-in by a CPU scheduler, can break at its first instruction.

- Any BRP (holding an IVA) can be linked with any other one with context ID capability. Several BRPs (holding IVAs) can be linked with the same context ID capable one.

- If a BRP (holding an IVA) is linked with one that is not configured for context ID comparison and linking, it is UNPREDICTABLE whether a Breakpoint Debug Event is generated or not. BCR[21:20] fields of the second BRP have to be set to 0b11.

- If a BRP (holding an IVA) is linked with one that is not implemented, it is UNPREDICTABLE whether a Breakpoint Debug Event is generated or not.

- If a BRP is linked with itself, it is UNPREDICTABLE whether a Breakpoint Debug Event is generated or not.

- If a BRP (holding an IVA) is linked with another BRP (holding a context ID value) and they are not both enabled (both BCR[0] bits set), the first one does not generate any Breakpoint Debug Event.

D2.3.7   Registers 96-111, Monitor debug-mode Value Registers (WVR)

Each WVR is associated with a WCR register: WVR0 with WCR0, WVR1 with WCR1, ..., WVR 15 with WCR15. These are WVRy and WCRy as defined in Table D2-1 on page D2-2. A pair of watchpoint registers WVRy/WCRy is called a *Watchpoint Register Pair* (WRP).

Copyright © 1996–2003 ARM Limited. All rights reserved.   ARM DDI 0100F-02
Confidential - Draft

The watchpoint value contained in this register always corresponds to a DVA. Watchpoints can be set either on a DVA or on a DVA/context ID pair. For the second case a WRP and a BRP with context ID comparison capability have to be linked (see *Registers 112-127, Watchpoint Control Registers (WCR)*). A Debug Event is generated when both the DVA and the context ID pair match at the same time. See Table D2-11 for details of WVR bits.

Table D2-11 WVR bit definition

Bits	Read/write attributes	Reset value	Description
[31:2]	RW	-	Watchpoint value

D2.3.8   Registers 112-127, Watchpoint Control Registers (WCR)

Each of these registers contain all the necessary control bits for setting appropriately either a simple watchpoint or a linked watchpoint, see Table D2-12.

Table D2-12 WVR bit definition

Bits	Read/write attributes	Reset value	Description
[0]	RW	0	Watchpoint enable. 0: watchpoint disabled. 1: watchpoint enabled.
[2:1]	RW	-	Supervisor access, see *Supervisor access, bits[2:1]*.
[4:3]	RW	-	Load/store access, see *Load/store access, bits[4:3]* on page D2-22.
[8:5]	RW	-	Byte address select, see *Byte address select, bits[8:5]* on page D2-23.
[15:9]	SBZ	-	RESERVED
[19:16]	RW	-	Linked BRP number.
[20]	RW	-	Enable linking.
[31:21]	UNP/SBZP	-	RESERVED

Supervisor access, bits[2:1]

The watchpoint can be conditional on the privilege of the access being done.

00   RESERVED.
01   Privileged.
10   User.
11   Either.

Copyright © 1996–2003 ARM Limited. All rights reserved.
Confidential - Draft

Load/store access, bits[4:3]

The watchpoint can be conditional on the type of access being done.

00   RESERVED.

01   Load.

10   Store.

11   Either.

———— **Note** ————

A `SWP` triggers on Load, Store or Either. A load exclusive instruction (`LDREX`) also triggers on Load. A store exclusive instruction (`STREX`) triggers on Store whether it succeeded or not.

————————————

Byte address select, bits[8:5]

The WVR is programmed with a word address. You can use this field to program the watchpoint so it hits only if certain byte addresses are accessed:

0000   The watchpoint never hits.

xxx1   The watchpoint hits if the byte at address BVR[31:2]+0 is accessed.

xx1x   The watchpoint hits if the byte at address BVR[31:2]+1 is accessed.

x1xx   The watchpoint hits if the byte at address BVR[31:2]+2 is accessed.

1xxx   The watchpoint hits if the byte at address BVR[31:2]+3 is accessed.

——— **Note** ———

These are little-endian byte addresses. This ensures that a watchpoint is triggered regardless of the way a memory position is accessed. For example, if a watchpoint is set on a byte in memory by setting WCR[8:5] = 0b0001, and if the word address is 0x0, then the watchpoint is triggered for both:

```
LDRB r0, #0x0
```

in little-endian mode, and:

```
LDRB r0, #0x3
```

in legacy big-endian mode (B bit of CP15 c1 set).

Linked BRP Number

The binary number encoded here indicates a contextID-holding BRP to link this WRP with.

Enable linking, bit[20]

0   Linking disabled.

1   Linking enabled.

When this bit is set this watchpoint is linked with the contextID-holding BRP selected by the Linked BRP Number field.

ARMv6 watchpoint debug event generation

This is how an ARMv6 Debug Architecture implementation must behave in generating Watchpoint Debug Events:

• The update of a WVR or a WCR can take effect a number of instructions after the corresponding MCR. It is only guaranteed to have taken effect by the next IMB.

• Any WRP can be linked with any BRP with context ID comparison capability. Several BRPs (holding IVAs) and WRPs can be linked with the same context ID capable BRP.

- If a WRP is linked with a BRP that is not configured for context ID comparison and linking, it is whether a Watchpoint Debug Event is generated or not. BCR[21:20] fields of the BRP have to be set to 0b11.

- If a WRP is linked with a BRP that is not implemented, it is UNPREDICTABLE whether a Watchpoint Debug Event is generated or not.

- If a WRP is linked with a BRP and they are not both enabled (BCR[0] and WCR[0] set), it does not generate any Watchpoint Debug Event.

D2.4 Reset values of the CP14 debug registers

Two different types of CP14 debug resets are considered:

System reset The reset generated by the activation of the main processor reset signal.

Debug logic reset A reset specific to the debug logic and generated through the External Debug Interface. If this interface complies with tbd xref [3] ARM Debug Interface, the debug logic reset is generated either by the **nDRST** signal or by the Debug-Logic-Reset DAPSM state.

On a debug logic reset, all the CP14 debug registers take the values indicated by the Reset value column in Table D2-5 on page D2-10, Table D2-7 on page D2-15, Table D2-8 on page D2-16, Table D2-9 on page D2-17, Table D2-10 on page D2-18, Table D2-11 on page D2-21, and Table D2-12 on page D2-21. DSCR[30:29] and DSCR[1:0] are special cases:

DSCR[1] Core Restarted

Not affected by a debug logic reset.

DSCR[0] Core Halted

Not affected by a debug logic reset.

DSCR[30] rDTRfull

Cleared on debug logic reset.

DSCR[29] wDTRfull

Set on debug logic reset.

Therefore, the core does not read the rDTR or write to the wDTR while in debug logic reset state. However, as soon as the debug logic reset state is abandoned, both these flags are initialized to 0.

On a system reset of the processor, all the CP14 debug registers keep their values. The exceptions are:

DSCR[0] Core Halted

A system reset forces the processor to leave Debug state. It always clears this flag.

DSCR[1] Core Restarted

A system reset forces the processor to leave Debug state. It always clears this flag.

D2.5 Access to CP14 debug registers from the external debug interface

All the CP14 debug registers that have been defined in this section must be accessible at the external debug interface. They must be accessible regardless of the processor state (ARM/Thumb/Java/Debug).

In tables above where an External view column is defined, those must be the read/write attributes at the external debug interface. Where there is only a Read/write column, those are the attributes for both the core and the external debugger.

——— **Note** ———

The ARM recommended specification of the External Debug Interface, the ARM Debug Interface, is documented separately from this manual.

Chapter D3
Debug Events and Exceptions

This chapter gives an introduction to the debug architecture. It contains the following sections:

- *Introduction* on page D3-2
- *Software debug events* on page D3-3
- *External debug* on page D3-4
- *Effect of a Debug Event on CP15 registers* on page D3-6
- *Debug exceptions* on page D3-7.

D3.1 Introduction

A Debug Event can be any of the following:

•   A Software Debug Event. See *Software debug events* on page D3-3.

•   The activation of the External Debug Request signal. See *External Debug Request signal* on page D3-4.

•   A Debug State Entry Request command. See *Debug State Entry Request command* on page D3-4.

D3.2 Software debug events

A software debug event can be any of the following:

- A Watchpoint Debug Event. This occurs when:
    — the DVA present in the data bus matches the watchpoint value
    — all the conditions of the WCR match
    — the watchpoint is enabled
    — the linked contextID-holding BRP (if any) is enabled and its value matches the context ID in CP15 register 13.

- A Breakpoint Debug Event. This occurs when:
    — an instruction was prefetched and the IVA present in the instruction bus matched the breakpoint value
    — at the same time the instruction was prefetched, all the conditions of the BCR matched
    — the breakpoint was enabled
    — at the same time the instruction was prefetched, the linked contextID-holding BRP (if any) was enabled and its value matched the Context ID in CP15 register 13
    — the instruction is now committed for execution.

- A Breakpoint Debug Event also occurs when:
    — an instruction was prefetched and the CP15 Context ID (register 13) matched the breakpoint value
    — at the same time the instruction was prefetched, all the conditions of the BCR matched
    — the breakpoint was enabled
    — the instruction is now committed for execution.

- A Software Breakpoint Debug Event. This occurs when a BKPT instruction is committed for execution.

- A Vector Trap Debug Event. This occurs when:
    — The instruction at a vector location was prefetched. This includes any kind of prefetches, not just the ones due to exception entry.
    — At the same time the instruction was prefetched, the corresponding bit of the VTR was set (vector trap enabled).
    — The instruction is now committed for execution.

D3.3 External debug

tbd

D3.3.1 External debug interface

An external debug interface is any set of external signals that an external debugger uses to access the debug resources of the core. The traditional External Debug Interface for ARM processors is based on the IEEE1149.1 standard. However, an ARMv6 implementation can use any other interface provided the following requirements are met

Regardless of the chosen interface, the following signals are mandatory:

DBGACK   Debug acknowledge signal. The processor asserts this output signal to indicate that the system has entered Debug state. See *Debug state* on page D3-8t for the definition of Debug state.

DBGEN   Debug enable signal. When this input signal is low (debug disabled), the processor behaves as if DSCR[15:14] equals 0b00, see tbd xref.

EDBGRQ   External debug request signal (). As described in tbd xref section 6.2 on page 22, this input signal forces the core into Debug state if the Debug logic is in Halted debug-mode.

——— **Note** ———

The ARM recommended specification of the External Debug Interface, the ARM Debug Interface, is documented separately from this manual.

D3.3.2 External Debug Request signal

An ARMv6 compliant core must have an External Debug Request input signal. This type of request can cause the processor to enter Debug state. If this happens, the DSCR[5:2] Method of Debug Entry bits is set to 0b0100.

This signal can be driven by the ETM to signal a trigger to the core. For example, if the processor is in Halted debug-mode and a memory permission fault occurs, an external Trace analyzer can collect trace information around this trigger event at the same time that the processor is stopped. Refer to the ARM Embedded Trace Macrocell Specification for more information.

An External Debugger can also drive this signal.

D3.3.3 Debug State Entry Request command

There must be a mechanism at the External Debug Interface whereby the core can be forced into Debug state. When this happens, the DSCR[5:2] Method of Debug Entry bits is set to 0b0000.

If the External Debug Interface adheres to the ARM Debug Interface, this mechanism is an IR instruction.

D3.3.4   Processor behavior on debug events

Table D3-1 shows how the processor behaves on Debug Events while not in Debug state.

When a Debug Event occurs and Halted debug-mode is selected and enabled then the processor enters Debug state.

A Debug exception is taken when Monitor Debug-Mode is selected and enabled, and a Software Debug Event occurs, with the following exceptions:

- Prefetch Abort, Data Abort, and Unaligned Address Vector Trap.

  These three Debug Events are ignored. This is to avoid the processor ending in an unrecoverable state on certain combinations of exceptions and vector traps.

- Unlinked context ID Breakpoint Debug Events, if the processor is running in a privileged mode.

- The External Debug Request signal and the Debug State Entry Request command.

When neither Halt nor Monitor Debug-Mode is selected and enabled, all Debug Events are ignored, except that the BKPT instruction generates a Prefetch Abort exception.

Table D3-1 Processor behavior on debug events

DSCR [15:14]	Mode (selected and enabled)	Action on Software Debug Event	Action on EDBGRQ	Action on Debug State Entry Request command
0b00	None	Ignore/PAbort [a]	Ignore	Ignore
0b01	Halt	Debug state entry	Debug state entry	Debug state entry
0b10	Monitor	Debug exception/Ignore [b]	Ignore	Ignore
0b11	Halt	Debug state entry	Debug state entry	Debug state entry

a. When debug is disabled, the BKPT instruction generates a Prefetch Abort exception instead of being ignored.
b. Prefetch Abort, Data Abort and Unaligned Address Vector Trap Debug Events are ignored in Monitor Debug-Mode. Unlinked context ID Breakpoint Debug Events are also ignored if the processor is running in a privileged mode and Monitor Debug-Mode is selected and enabled.

D3.4 Effect of a Debug Event on CP15 registers

A Debug Event has the following effect on CP15 registers:

Halted debug-mode entry

- writes to the DPSR and FAR are disabled
- The WFAR is updated with an address related to the instruction causing the watchpoint.

Monitor debug-mode entry

- the contents of the FAR are UNPREDICTABLE
- the WFAR is updated with an address related to the instruction causing the watchpoint
- the DFSR is updated with the Debug Event code (bit[10] == 0, bits[3:0] == 0x0100)

Normal Data Abort

- the contents of the WFAR are UNPREDICTABLE (use R14_abt register)
- the FAR relates to the address of the failing access (it may not be the actual address in unaligned cases)
- the DFSR is updated with the abort reason.

D3.5 Debug exceptions

A Debug exception is taken when Monitor Debug-Mode is enabled, and a Software Debug Event occurs, with the following exceptions:

- Prefetch Abort, Data Abort, and Unaligned Address Vector Trap.

  These three Debug Events are ignored. This is to avoid the processor ending in an unrecoverable state on certain combinations of exceptions and vector traps.

- Unlinked context ID Breakpoint Debug Events, if the processor is running in a privileged mode.

- The External Debug Request signal and the Debug State Entry Request command.

If the cause of the Debug exception was a Watchpoint Debug Event, the processor performs the following actions:

1. The DSCR[5:2] Method of Debug Entry bits are set to Watchpoint occurred.

2. The CP15 DFSR, FAR and WFAR registers are set as described in *Effect of a Debug Event on CP15 registers* on page D3-6.

3. The same sequence of actions as in a Data Abort exception is performed. This includes setting the R14_abt, base register and destination registers to the same values as if this was a Data Abort.

The Data Abort handler is responsible for checking the DFSR or DSCR[5:2] bits to find out whether the routine entry was caused by a Debug exception or a Data Abort exception. If the cause was a Debug exception, it must branch to the Monitor Target. The watchpointed address can be found in the FAR, the address of the instruction that caused the Watchpoint Debug Event in the WFAR and the address of the instruction to restart at, plus 0x08, in R14_abt.

If the cause of the Debug exception was a Breakpoint, Software Breakpoint or Vector Trap Debug Event, the processor performs the following actions:

- The DSCR[5:2] Method of Debug Entry bits are set appropriately.

- The CP15 IFSR register is set as described in *Effect of a Debug Event on CP15 registers* on page D3-6.

- The same sequence of actions as in a Prefetch Abort exception is performed.

The Prefetch Abort handler is responsible for checking the IFSR or DSCR[5:2] bits to find out whether the routine entry was caused by a Debug exception or a Prefetch Abort exception. If the cause was a Debug exception, it should branch to the Monitor Target. The address of the instruction causing the Software Debug Event, plus 0x04, can be found in R14_abt.

D3.6   Debug state

When the conditions stated in *Processor behavior on debug events* on page D3-5 are met then the processor switches to a special state called *Debug state*. While in Debug state, the processor must behave as follows:

*   The DSCR[0] Core Halted bit is set.

*   The **DBGACK** signal (see *External debug* on page D3-4) is asserted.

*   The DSCR[5:2] Method of Debug Entry bits are set appropriately.

*   The CP15 FAR and WFAR registers are set as described in *Effect of a Debug Event on CP15 registers* on page D3-6. The IFSR and DFSR registers are not modified.

*   The processor is halted. The pipeline is flushed and no instructions are prefetched.

*   The processor does not change the execution mode. The CPSR is not altered.

*   The DMA engine keeps running. The External Debugger can stop it and restart it using CP15 operations. See *L1 DMA control using CP15 Register 11* on page B7-9 for details.

*   Interrupts and exceptions are treated as described in *Interrupts in Debug state* on page D3-10 and *Exceptions* on page D3-10.

*   Software Debug Events are ignored.

*   The External Debug Request signal is ignored.

*   Debug State Entry Request commands are ignored.

*   There must be a mechanism, via the External Debug Interface, whereby the core can be forced to execute an ARM state instruction. This mechanism is enabled through DSCR[13] Execute ARM Instruction enable bit.

    The core executes the instruction as in ARM state, regardless of the actual value of the T and J bits of the CPSR. If both the J bit and T bit are set, the behavior is UNPREDICTABLE.

*   In this state the core can execute any ARM state instruction, as if in a privileged mode. For example, if the processor is in User mode then the MSR instruction is allowed to update the PSRs and all the CP14 debug instructions are allowed to execute. However, the processor still accesses the register bank and memory as indicated by the CPSR Mode bits. For example, if the processor is in User mode then it sees the User mode register bank and accesses the memory without any privilege.

*   The PC behaves as described in *Behavior of the PC in Debug state* on page D3-9.

There must be a mechanism, a restart command, that the External Debugger can use to force the processor out of Debug state. This restart command must clear the DSCR[1] Core Restarted flag. When the processor has actually exited Debug state, the DSCR[1] Core Restarted bit must be set and the DSCR[0] Core Halted bit and **DBGACK** signal must be cleared.

D3.6.1   Behavior of the PC in Debug state

- The PC is frozen on entry to Debug state, that is, it does not increment on ARM instructions execution. However, branches and instructions that modify the PC directly do update it.

- If the PC is read after the processor has entered Debug state, it returns a value as described in Table D3-2, depending on the previous state and the type of Debug Event.

- If a sequence for writing a certain value to the PC is executed while in Debug state, and then the processor is forced to restart, the execution starts at the address corresponding to the written value. However, the CPSR must be set to the return state (ARM/Thumb/Java) before the PC is written to. Otherwise, the processor behavior is UNPREDICTABLE.

- If the processor is forced to restart without having performed a write to the PC, the restart address is UNPREDICTABLE.

- If the PC or the CPSR is written to while in Debug state, subsequent reads to the PC returns an UNPREDICTABLE value.

- If an instruction that writes to the PC fails its condition codes then the PC will be written to with an UNPREDICTABLE value. That is, if the processor is then forced to restart, or if the PC is read, the results are UNPREDICTABLE.

——— **Note** ———

Implementations that make use of branch prediction may not be able to easily stop the PC from changing when the branch is predicted, taken, and shortly after recovers to the next sequential address.

Table D3-2 Read PC value after Debug state entry

Debug event	ARM	Thumb	Java [a]	Return address meaning [b]
Breakpoint	RA+8	RA+4	RA + offset	Breakpointed instruction address
Watchpoint	RA+8	RA+4	RA + offset	Address of the instruction for the execution to resume [c]
BKPT instruction	RA+8	RA+4	RA + offset	BKPT instruction address
Vector catch	RA+8	RA+4	RA + offset	Vector address
EDBGRQ signal	RA+8	RA+4	RA + offset	Address of the instruction for the execution to resume
Debug state entry request command	RA+8	RA+4	RA + offset	Address of the instruction for the execution to resume

a. offset is an implementation defined constant and documented value.
b. Return address is the address of the instruction that the processor should first execute on Debug state exit.
c. Watchpoints can be imprecise. The return address is not the address of the instruction just after the one that hit the watchpoint, the processor might stop a number of instructions later. The address of the instruction that hit the watchpoint can be found in the CP15 WFAR.

D3.6.2   Interrupts in Debug state

Interrupts are ignored in Debug state regardless of the value of the I and F bits of the CPSR. These bits are not be changed because of the Debug state entry.

D3.6.3   Exceptions

Reset, Prefetch, Debug, SWI, and Undefined exceptions are treated as follows in Debug state:

Reset Taken as in a normal processor state (ARM/Thumb/Java). The processor leaves Debug state as a result of the system reset.

Prefetch Abort Cannot occur because no instructions are prefetched in Debug state.

Debug Cannot occur because Software Debug Events are ignored while in Debug state.

SWI Executing a `SWI` while in Debug state results in UNPREDICTABLE behavior.

Undefined Executing an undefined instruction while in Debug state results in UNPREDICTABLE behavior.

All other exceptions are taken, as explained in *Taking an exception in Debug state*.

Taking an exception in Debug state

When an exception is taken while in Debug state, the behavior of the core must be as follows:

*   The PC, CPSR, SPSR_<exception_mode> are set in the same way as in a normal processor state exception entry. If the exception is an Imprecise Data Abort, and the PC has not yet been written, R14_abt is set as in a normal processor state exception entry, In all other cases, R14_<exception_mode> is set to an UNPREDICTABLE value.

*   The processor remains in Debug state. Because it is in Debug state, it does not prefetch the exception vector.

In addition, if the exception is a Data Abort:

*   The DFSR and FAR are set in the same way as in a normal processor state exception entry. The WFAR is set to an UNPREDICTABLE value.

    —— **Note** ——

    The IFSR will never be modified because no prefetch abort can occur while in Debug state.

*   The DSCR[6] Sticky Abort bit is set.

*   The DSCR[5:2] Method of Debug Entry are set to D-side abort occurred. The I-side cannot miss because no instruction prefetches are done in Debug state.

ARM DDI 0100F-02

Exception priorities in Debug state

Table D3-3 shows the exception priorities in Debug state.

Table D3-3 Exception priorities in Debug state

Priority	Exception
Highest	Reset
	Watchpoint Debug Event
	FIQ
	IRQ
	Debug State Entry Request command
	External Debug Request signal
	Prefetch Abort
	Breakpoint Debug Event, Vector Trap Debug Event
Lowest	Software Breakpoint Debug Event (BKPT), SWI, Undefined Instruction, Coprocessor Undefined Instruction

Glossary

Abort   Is caused by an illegal memory access. Aborts can be caused by the external memory system or the MMU.

Abort model

Describes what happens to the processor state when a Data Abort exception occurs. Different abort models behave differently with regard to load/store instructions that specify base register write-back. For more details, see *Effects of data-aborted instructions* on page A2-21.

Addressing modes

Generally mean a procedure shared by many different instructions, for generating values used by the instructions. For four of the ARM addressing modes, the values generated are memory addresses (which is the traditional role of an addressing mode). A fifth addressing mode generates values to be used as operands by data-processing instructions.

Aligned   Refers to data items stored in such a way that their address is divisible by the highest power of 2 that divides their size. Aligned halfwords, words and doublewords therefore have addresses that are divisible by 2, 4 and 8 respectively.

AL (always)

Specifies that the instruction is executed irrespective of the value of the condition code flags. If no condition code is given with an instruction mnemonic, the AL condition code is used.

ALU   Stands for Arithmetic Logic Unit.

AND   Performs a bitwise AND.

Arithmetic_Shift_Right

Performs a right shift, repeatedly inserting the original left-most bit (the sign bit) in the vacated bit positions on the left.

ARM instruction

Is a word which specifies an operation for an ARM processor to perform. ARM instructions must be word-aligned.

Assert statements

Are used in pseudo-code to indicate that a certain condition has been met.

Assignment

Is signified by =.

Banked registers

Are register numbers whose physical register is defined by the current processor mode. The banked registers are registers R8 to R14.

Base register

Is a register specified by a load/store instruction that is used as the base value for the instruction's address calculation. Depending on the instruction and its addressing mode, an offset can be added to or subtracted from the base register value to form the virtual address which is sent to memory.

Base register write-back

Is when the base register used in the address calculation has a modified value written to it.

Big-endian memory

Means that:

•   a byte or halfword at a word-aligned address is the most significant byte or halfword within the word at that address

•   a byte at a halfword-aligned address is the most significant byte within the halfword at that address.

Binary numbers

Are preceded by 0b.

Blocking

Blocking The Cache block transfer operations for cleaning and/or invalidating a range of addresses from the cache are described as blocking operations in that following instructions must not be executed while this operation is in progress.

A non-blocking operation can permit following instructions to be executed before the operation is completed, and in the event of encountering an exception do not signal an exception to the core. This allows implementations to retire following instructions while the non-blocking operation is executing, without the need to retain precise processor state.

Boolean AND

Is signified by the AND operator.

Boolean OR

Is signified by the OR operator.

ARM DDI 0100F-02

BorrowFrom

Returns 1 if the subtraction specified as its parameter caused a borrow (the true result is less than 0, where the operands are treated as unsigned integers), and returns 0 in all other cases. This delivers further information about a subtraction which occurred earlier in the pseudo-code. The subtraction is not repeated.

Branch prediction

Is where an ARM implementation chooses a future execution path to prefetch along (see Prefetching). For example, after a branch instruction, the implementation can choose to prefetch either the instruction following the branch or the instruction at the branch target.

Byte   Is an 8-bit data item.

Byte-invariant

A method of switching between little-endian and big-endian operation that leaves byte accesses unchanged. Accesses to other data sizes are necessarily affected by such endianness switches.

Cache   Is a block of high-speed memory locations whose addresses are changed automatically in response to which memory locations the processor is accessing, and whose purpose is to increase the average speed of a memory access.

Cache contention

Is when the number of frequently-used memory cache lines that use a particular cache set exceeds the set-associativity of the cache. In this case, main memory activity goes up and performance drops.

Cache hit

Is a memory access which can be processed at high speed because the data it addresses is already in the cache.

Cache line

Is the basic unit of storage in a cache. Its size is always a power of two (usually 4 or 8 words), and is required to 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 just called cache lines.

Cache line index

Is a number associated with each cache line in a cache set. Within each cache set, the cache lines are numbered from 0 to (set associativity)–1.

Cache lockdown

Alleviates the delays caused by accessing a cache in a worst-case situation. Cache lockdown allows critical code and data to be loaded into the cache so that the cache lines containing them are not subsequently re-allocated. This ensures that all subsequent accesses to the code and data concerned are cache hits and so complete quickly.

Cache lockdown blocks

Consist of one line from each cache set. Cache lockdown is performed in units of a cache lockdown block.

Cache miss

Is a memory access which cannot be processed at high speed because the data it addresses is not in the cache.

Cache sets

Are areas of a cache, 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.

Cache way

A cache way consists of one cache line from each cache set. The cache ways are indexed from 0 to ASSOCIATIVITY-1. The cache lines in a cache way are chosen to have the same index as the cache way. So for example cache way 0 consists of the cache line with index 0 from each cache set, and cache way *n* consists of the cahse line with index *n* from each cache set.

Callee-save registers

Are registers that a called procedure must preserve. To preserve a callee-save register, the called procedure would normally either not use the register at all, or store the register to the stack during procedure entry and re-load it from the stack during procedure exit.

Caller-save registers

Are registers that a called procedure need not preserve. If the calling procedure requires their values to be preserved, it must store and reload them itself.

CarryFrom

Returns 1 if the addition specified as its parameter caused a carry (true result is bigger than $2^{32}-1$, where the operands are treated as unsigned integers), and returns 0 in all other cases. This delivers further information about an addition which occurred earlier in the pseudo-code. The addition is not repeated.

case ... endcase statements

Are used to indicate a one of many execution option. Indentation indicates the range of statements in each option.

ClearByAddress(<physical_address>)

Clears any request by any processor to mark address `<physical_address>` as exclusive access.

ClearByProcessor(<processor_id>)

Clears any request by processor `<processor_id>` to mark any address as exclusive access.

ClearOthersByAddress(<physical_address>,<processor_id>)

Clears any request by any processor, other than processor `<processor_id>`, to mark address `<physical_address>` as exclusive access.

Comments

Are enclosed in /* */.

Condition field

Is a 4-bit field in an instruction that is used to specify a condition under which the instruction can execute.

Conditional execution

Means that if the condition code flags indicate that the corresponding condition is true when the instruction starts executing, it executes normally. Otherwise, the instruction does nothing.

ConditionPassed(cond)

Returns TRUE if the state of the N, Z, C and V flags fulfils the condition encoded in the cond argument, and returns FALSE in all other cases.

Configuration

Settings made on reset, or immediately after reset, and normally expected to remain static throughout program execution.

Control bits

    Are the bottom eight bits of a *Program Status Register* (PSR). The control bits change when an exception arises and can be altered by software only when the processor is in a privileged mode.

CPSR   Is the Current Program Status Register.

CurrentModeHasSPSR()

    Returns TRUE if the current processor mode is not User mode or System mode, and returns FALSE if the current mode is User mode or System mode.

Data cache

    Is a separate cache used only for processing data loads and stores.

Decode bits

    Are bits[27:20] and bits[7:4] of an ARM instruction, and are the main bits used to determine the type of instruction to be executed.

Digital signal processing

    Refers to a variety of algorithms which are used to process signals that have been sampled and converted to digital form. Saturated arithmetic is often used in such algorithms.

Direct-mapped cache

    Is a one-way set-associative cache. Each cache set consists of a single cache line, so cache look-up just needs to select and check one cache line.

Direct Memory Access

    Is an operation that accesses main memory directly, without the processor performing any accesses to the data concerned.

Domain   Is a collection of sections, large pages and small pages of memory, which can have their access permissions switched rapidly by writing to the Domain Access Control Register (CP15 register 3).

Do-not-modify fields (DNM)

    Means the value must not be altered by software. DNM fields read as UNPREDICTABLE values, and can only be written with the same value read from the same field on the same processor.

    Throughout this manual, DNM fields are sometimes followed by RAZ or RAO in parentheses as a guideline to implementors as to which way the bits should read for future compatibility, but programmers must not rely on this behavior.

Double-precision value

    Consists of two 32-bit words which must appear consecutively in memory and must both be word-aligned, and which is interpreted as a basic double-precision floating-point number according to the IEEE 754-1985 standard.

Doubleword

    Is 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

Elements

Are separated by | in a list of possible values for a variable.

Endianness

is an aspect of the system's memory mapping. See big-endian and little-endian.

EOR   Performs a bitwise Exclusive OR.

Exception

Handles an event. For example, an exception could handle an external interrupt or an undefined instruction.

Exception modes

Are privileged modes that are entered when specific exceptions occur.

Exception vector

Is one of a number of fixed addresses in low memory, or in high memory if high vectors are configured.

ExecutingProcessor()

Returns a value corresponding to the processor executing the operation. Each processor in the system has a distinct value.

Explicit access

A read from memory, or a write to memory, generated by a load or store instruction executed in the CPU. Reads and writes generated by L1 DMA accesses or hardware pagetable accesses are not explicit accesses.

External abort

Is an abort that is generated by the external memory system.

Fault   Is an abort that is generated by the MMU.

FCSE (Fast Context Switch Extension)

Modifies the behavior of an ARM memory system to allow multiple programs running on the ARM processor to use identical address ranges, while ensuring that the addresses they present to the rest of the memory system differ.

Flat address mapping

Is where the physical address for every access is equal to its virtual address.

Flush-to-zero mode

Is a special processing mode that optimizes the performance of some VFP algorithms by replacing the denormalized operands and intermediate results with zeros, without significantly affecting the accuracy of their final results.

Floating-point Exception Register

Is a read/write register, two bits of which provide system-level status and control. The remaining bits of this register can be used to communicate exception information between the hardware and software components of the implementation, in an IMPLEMENTATION DEFINED manner.

Floating-point Status and Control Register

Is a read/write register which provides all user-level status and control of the floating-point system.

Floating-point System ID Register

Is a read-only register whose value indicates which VFP implementation is being used.

ARM DDI 0100F-02

for ... statements
> Are used to indicate a loop over a numeric range. Indentation is used to indicate the range of statements in the loop.

FPEXC   *See* Floating-point Exception Register.

FPSCR   *See* Floating-point Status and Control Register.

FPSID   *See* Floating-point System ID Register.

Fully-associative cache
> Has just one cache set, which consists of the entire cache. See also direct-mapped cache.

General-purpose register
> Is one of the 32-bit general-purpose integer registers, R0 to R15. Note that R15 holds the Program Counter, and there are often limitations on its use that do not apply to R0 to R14.

Halfword
> Is a 16-bit data item. Halfwords are normally halfword-aligned in ARM systems.

Halfword-aligned
> Means that the address is divisible by 2.

Hexadecimal numbers
> Are preceded by `0x` and are given in a monospaced font.

High registers
> Are ARM registers 8 to 15, which can be accessed by some Thumb instructions.

High vectors
> Are alternative locations for exception vectors. The high vector address range is near the top of the address space, rather than at the bottom.

if ... else if ... else statements
> Are used to signify conditional statements. Indentation indicates the range of statements in each option.

IGNORE fields (IGN)
> Must ignore writes.

IMB   *See* Instruction Memory Barrier.

Instruction Memory Barrier
> Is a sequence of operations that can be used in the middle of a self-modifying code sequence to make it execute reliably. This sequence often depends both on the ARM processor implementation and on the memory system implementation.

Immediate and offset fields
> Are unsigned unless otherwise stated.

Immediate values
> Are values which are encoded directly in the instruction and used as numeric data when the instruction is executed. Many ARM and Thumb instructions allow small numeric values to be encoded as immediate values within the instruction that operates on them.

IMP   Is an abbreviation used in diagrams to indicate that the bit or bits concerned have IMPLEMENTATION DEFINED behavior.

IMPLEMENTATION DEFINED fields
Means that the behavior is not architecturally defined, but should be defined and documented by individual implementations.

InAPrivilegedMode()
Returns TRUE if the current processor mode is not User mode, and returns FALSE if the current mode is User mode.

Index register
Is a register specified in some load/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 virtual address which is sent to memory. Some addressing modes optionally allow the index register value to be shifted prior to the addition or subtraction.

Inline literals
These are constant addresses and other data items held in the same area as the code itself. They are automatically generated by compilers, and can also appear in assembler code.

Instruction cache
Is a separate cache used only for processing instruction fetches.

Instruction memory barrier
An IMPLEMENTATION DEFINED sequence of operations that ensure that all following instructions are fetched and executed after the effects of all previous instructions have completed.

Interworking
Is a method of working that allows branches between ARM and Thumb code.

IsExclusiveGlobal(<physical_address>,<processor_id>)
Returns TRUE if the processor `<processor_id>` has marked the address `<physical_address>` as exclusive access requested. Otherwise, it returns FALSE.

IsExclusiveLocal(<processor_id>)
Returns TRUE if the processor `<processor_id>` has marked anaddress as exclusive access requested. Otherwise, it returns FALSE.

Little-endian memory
Means that:

• a byte or halfword at a word-aligned address is the least significant byte or halfword within the word at that address

• a byte at a halfword-aligned address is the least significant byte within the halfword at that address.

Load/Store architecture
Is an architecture where data-processing operations only operate on register contents, not directly on memory contents.

ARM DDI 0100F-02

Logical_Shift_Left

Performs a left shift, inserting zeros in the vacated bit positions on the right. `<<` is used as a short form for `Logical_Shift_Left`.

Logical_Shift_Right

Performs a right shift, inserting zeros in the vacated bit positions on the left.

Long branch

Is the use of a load instruction to branch to anywhere in the 4GB address space.

LR (Link Register)

Is integer register R14.

MarkExclusiveGlobal(<physical_address>,<processor_id>)

Records the fact that processor `<processor_id>` has requested exclusive access to address `<physical_address>`. This causes any previous request for exclusive access by the same processor to any other address to be cleared.

MarkExclusiveLocal(<processor_id>)

Records the fact that processor `<processor_id>` has requested exclusive access to an address.

Memory[<address>,<size>]

Refers to a data item in memory of length `<size>`, at address `<address>`, aligned on a `<size>` byte boundary. The data item is zero-extended to 32 bits. Currently defined sizes are:

1 for bytes

2 for halfwords

4 for words

To align on a `<size>` boundary, halfword accesses ignore `<address>`[0] and word accesses ignore `<address>`[1:0].

Memory barrier

An IMPLEMENTATION DEFINED sequence of operations that ensure that all explicit accesses occurring in program order before the barrier are executed before the operation that acts as the barrier, and that all explicit accesses occurring in program order after the barrier are executed after the barrier.

Memory coherency

Is the problem of ensuring that when a memory location is read (either by a data read or an instruction fetch), the value 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, a write buffer and/or cache(s).

Memory Management Unit

Allows detailed control of a memory system. Most of the control is provided via translation tables held in memory.

Memory-mapped I/O

Uses special memory addresses which supply I/O functions when they are loaded from or stored to.

Mixed-endian

A processor supports mixed-endian memory accesses if accesses to big-endian data and little-endian data can be freely intermixed, with only small performance and code size penalties for doing so.

Modified Virtual Address

Is the address produced by the FCSE which is sent to the rest of the memory system to be used in place of the normal virtual address. Use of the FCSE is deprecated in new designs.

MMU   *See* Memory Management Unit.

MVA   *See* Modified Virtual Address.

NaN   Means Not a Number, and is a type of floating-point value.

NOT   Performs a bitwise complement.

NotFinished(CP_number)

Returns TRUE if the coprocessor signified by the `CP_number` argument has signalled that the current operation is incomplete, and returns FALSE if the operation is complete.

NumberOfSetBitsIn(bitfield)

Performs a population count on (counts the set bits in) the bitfield argument.

Object[from:to]

Indicates the bit field extracted from Object starting at bit "from", ending with bit "to" (inclusive)

Offset addressing

Means that the memory address is formed by adding or subtracting an offset to or from the base register value.

Optional parts of instructions

Are surrounded by { and }.

OR   Performs a bitwise Inclusive OR.

OverflowFrom

Returns 1 if the addition or subtraction specified as its parameter caused a 32-bit signed overflow. Addition generates an overflow if both operands have the same sign (bit[31]), and the sign of the result is different to the sign of both operands. Subtraction causes an overflow if the operands have different signs, and the first operand and the result have different signs.

This delivers further information about an addition or subtraction which occurred earlier in the pseudo-code. The addition or subtraction is not repeated.

PC (Program Counter)

Is integer register R15.

PCB (Process Control Block)

In software systems that support multiple software processes, is a data structure associated with each process that holds the process's state while it is not executing.

Physical address

Identifies a main memory location.

Predictable subsequent execution

Means execution of any instructions that can be reached subsequently by any combination of normal sequential execution and executing branches with statically-determined targets. Any instruction which branches to a location which depends on register values (such as MOV PC, LR) terminates predictable subsequent execution

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.

Prefetching

Is the process of fetching instructions from memory before the instructions that precede them have finished executing. Prefetching an instruction does not mean that the instruction has to be executed.

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.

Privileged mode

Is any processor mode other than User mode. Memory systems typically check memory accesses from privileged modes against supervisor access permissions rather than the more restrictive user access permissions. The use of some instructions is also restricted to privileged modes.

Process ID

In the FCSE, this is a 7-bit number that identifies which process block the current process is loaded into.

Protection region

Is a memory range whose position, size, and other properties are defined by Protection Unit registers.

Protection Unit

Is a hardware unit whose registers provide simple control of a limited number of protection regions in memory.

PSR Is the CPSR or one of the SPSRs.

Quiet NaN

Is a NaN that propagates unchanged through most floating-point operations.

Read-allocate cache

Is a cache in which a cache miss on storing data causes the data to be written to main memory. Cache lines are only allocated to memory locations when data is read/loaded, not when it is written/stored.

Read-As-Zero fields (RAZ)

Appear as zero when read.

Read-Modify-Write fields (RMW)

Are read to a general-purpose register, the relevant fields updated in the register, and the register value written back.

RISC Stands for Reduced Instruction Set Computer.

Rotate_Right

Performs a right rotate, where each bit that is shifted off the right is inserted on the left.

Rounding error

Is defined to be the value of the rounded result of an arithmetic operation minus the exact result of the operation.

Rounding modes

Specify how the exact result of a floating-point operation is rounded to a value which is representable in the destination format.

Round to Nearest (RN) mode

Means that the rounded result is the nearest representable number to the unrounded result.

Round towards Plus Infinity (RP) mode

Means that the rounded result is the nearest representable number which is greater than or equal to the exact result.

Round towards Minus Infinity (RM) mode

Means that the rounded result is the nearest representable number which is less than or equal to the exact result.

Round towards Zero (RZ) mode

Means that results are rounded to the nearest representable number which is no greater in magnitude than the unrounded result.

Saturated arithmetic

Is 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 vice versa.

Security hole

Is an illegal mechanism that bypasses system protection.

Self-modifying code

Is code which writes one or more instructions to memory and then executes them. This type of code cannot be relied on without the use of an IMB.

Set-associativity

Is the number of cache lines in each of the cache sets in a cache. It can be any number ≥ 1, and is not restricted to being a power of two.

Shared(<Rm>)

If CP15 register 1 bit[0] (Mbit) is set, returns the value of the shared memory region attribute corresponding to the virtual address in Rm for the current process ID and TLB entries for the current processor. If Mbit is not set, it returns 0.

Shifter operand

Is one of the source operands of an ARM data-processing instruction. It is either an immediate value or a register.

Should-Be-One fields (SBO)

Should be written as 1 (or all 1s for bit fields) by software. Values other than 1 produce UNPREDICTABLE results.

Should-Be-One-or-Preserved fields (SBOP)

Should be written as 1 (or all 1s for bit fields) or preserved by writing the same value that has been previously read from the same fields on the same processor.

Should-Be-Zero fields (SBZ)

Should be written as zero (or all 0s for bit fields) by software. Non-zero values produce UNPREDICTABLE results.

Should-Be-Zero-or-Preserved fields (SBZP)

Should be written as zero (or all 0s for bit fields) or preserved by writing the same value that has been previously read from the same fields on the same processor.

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.

SignedDoesSat(x,n)

Returns 0 if x lies inside the range of an n-bit signed integer (that is, if $-2^{(n-1)} \le x \le 2^{(n-1)} - 1$), and 1 otherwise.

This operation delivers further information about a SignedSat(x, n) operation which occurred earlier in the pseudo-code. Any operations used to calculate x or n are not repeated.

SignExtend(arg)

Sign-extends (propagates the sign bit) its argument to 32 bits.

SignedSat(x,n)

Returns x saturated to the range of an n-bit signed integer. That is, it returns:

- $-2^{(n-1)}$ if $x < -2^{(n-1)}$

- x if $-2^{(n-1)} <= x <= 2^{(n-1)} - 1$

- $2^{(n-1)} - 1$ if $x > 2^{(n-1)} - 1$.

SIMD Means Single-Instruction, Multiple-Data operations.

Single-precision value

Is a 32-bit word, and must be word-aligned when held in memory, and which is interpreted as a basic single-precision floating-point number according to the IEEE 754-1985 standard.

SP (Stack Pointer)

    Is integer register R13.

Spatial locality

    Is 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.

SPSR Is the Saved Program Status Register which is associated with the current processor mode (and is undefined if there is no such Saved Program Status Register, as in User mode or System mode).

SWI Is a software interrupt.

Status registers

    See CPSR and SPSR.

Tag bits Are bits[31:L+S]) of a virtual address, where L and S are the logarithms base 2 of the cache line length and the number of cache sets respectively. A cache hit occurs if the tag bits of the virtual address supplied by the ARM processor match the tag bits associated with a valid line in the selected cache set.

Temporal locality

    Is 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.

Test for equality

    Is signified by ==.

Thumb instruction

    Is a halfword which specifies an operation for an ARM processor in Thumb state to perform. Thumb instructions must be halfword-aligned.

TLB *See* Translation Lookaside Buffer.

TLB lockdown

    Is a way to prevent specific translation table walk results being accessed. This ensures that accesses to the associated memory areas never cause a translation table walk.

TLB(<Rm>)

    If CP15 register 1 bit[0] (Mbit) is set, returns the physical address corresponding to the virtual address in Rm, for the current process ID and TLB entries for the executing processor. If Mbit is not set, it returns the value in Rm.

Translation Lookaside Buffer

    Is a memory structure containing the results of translation table walks. They help to reduce the average cost of a memory access. Usually, there is a TLB for each memory interface of the ARM implementation.

Translation tables

    Are tables held in memory. They define the properties of memory areas of various sizes from 1KB to 1MB.

Translation table walk

    Is the process of doing a full translation table lookup. It is performed automatically by hardware.

ARM DDI 0100F-02

Trap enable bits

> Determine whether trapped or untrapped exception handling is selected. If trapped exception handling is selected, the way it is carried out is IMPLEMENTATION DEFINED.

Unaffected items

> Are not changed by a particular operation.

Unaligned memory accesses

> Are memory accesses that are not, or might not be, appropriately halfword-aligned, word-aligned, or doubleword aligned.

Unbanked registers

> Are general-purpose registers that refer to the same 32-bit physical register in all processor modes. Unbanked registers are registers R0 to R7.

UNDEFINED

> Indicates an instruction that generates an Undefined Instruction exception. See *Undefined Instruction exception* on page A2-18 for information on Undefined Instruction exceptions.

Unified cache

> Is a cache used for both processing instruction fetches and processing data loads and stores.

Unindexed addressing

> Indicates addressing in which the base register value is used directly as the virtual address to send to memory, without adding or subtracting an offset. In most types of addressing mode, unindexed addressing is performed by using offset addressing with an immediate offset of 0. ARM Addressing Mode 5 (used for LDC and STC instructions) has an explicit unindexed addressing mode which allows the offset field in the instruction to be used to specify additional coprocessor options.

UNPREDICTABLE

> Means the result of an instruction cannot be relied upon. UNPREDICTABLE instructions or results must not represent security holes. UNPREDICTABLE instructions must not halt or hang the processor, or any parts of the system.

UNPREDICTABLE fields (UNP)

> Do not contain valid data, and a value can vary from moment to moment, instruction to instruction, and implementation to implementation.

Unsigned data types

> Represent a non-negative integer in the range 0 to $+2^N-1$, using normal binary format.

UnsignedDoesSat(x,n)

> Returns 0 if x lies within the range of an n-bit unsigned integer (that is, if $0 \le x < 2^n$) and 1 otherwise.
>
> This operation delivers further information about an UnsignedSat(x,n) operation that occurred earlier in the pseudo-code. Any operations used to calculate x or n are not repeated.

UnsignedSat(x,n)

> Returns x saturated to the range of an n-bit unsigned integer. That is, it returns:
>
> • 0 if x < 0

- x if $0 <= x < 2^n$

- $2^n - 1$ if $x > 2^n - 1$

Variable parts of instructions

Are surrounded by < and >.

VFP *See* Vector Floating-point Architecture.

Vector Floating-point Architecture

Is a coprocessor extension to the ARM architecture. It provides single-precision and double-precision floating-point arithmetic.

VFP emulator

Is an implementation which consists of software only, with all floating-point arithmetic being emulated by ARM routines.

Virtual address

Is an address generated by an ARM processor.

while .... statements

Are used to indicate a loop. Indentation indicates the range of statements in the loop.

Word Is a 32-bit data item. Words are normally word-aligned in ARM systems.

Word-aligned

Means that the address is divisible by 4.

Word-invariant

A way of switching between little-endian and big-endian operation that leaves aligned word accesses unchanged. Accesses to other data sizes and to unaligned words are necessarily affected by such endianness switches.

Write-allocate cache

Is a cache in which a cache miss on storing data causes a cache line to be allocated and main memory contents to be read into it, followed by writing the stored data into the cache line.

Write-back cache

Is 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 re-allocated. Another common term for a write-back cache is a *copy-back cache*.

Write-through cache

Is 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 via a write buffer, to avoid slowing down the processor.

Write buffer

Is a block of high-speed memory whose purpose is to optimize stores to main memory.

ARM DDI 0100F-02

{msbyte,..lsbyte}
> Byte concatenation operation with least significant byte at the right bus[msbit:lsbit] Sub-bus nomenclature, with field bits denoted by msbit down to lsbit.

b3...b0   Lower case b prefix indicates little-endian byte interpretation.

B3...N0   Upper case B prefix indicates big-endian byte interpretation.

Glossary

ARM DDI 0100F-02

Index

The items in this index are listed in alphabetical order, with symbols and numerics appearing at the end. The references given are to page numbers.

Invalid Operation exceptions   C2-5,
  C2-10
IRQ (interrupt request)   A2-3

J

Java
  J bit in PSR   A2-12, A2-13

L

LDC instruction   A4-33
LDC2 instruction   A4-33
LDM instruction
  ARM   A4-37
LDMIA instruction
  Thumb   A7-45
LDR instruction
  ARM   A4-42
  Thumb   A7-50
LDRB instruction
  ARM   A4-45
  Thumb   A7-55
LDRBT instruction   A4-47
LDRD instruction   A4-49, A4-51,
  A4-208
LDRH instruction
  ARM   A4-54
  Thumb   A7-56
LDRSB instruction
  ARM   A4-55
  Thumb   A7-60
LDRSH instruction
  ARM   A4-57
  Thumb   A7-61
LDRT instruction   A4-59
Link register (LR)   A1-3, A1-6
list of instructions in   A7-121
Load and Store
  addressing modes
    offset   A1-8
    post-indexed   A1-9
    pre-indexed   A1-8
  doublewords   A4-49, A4-51,
    A4-206, A4-208
  instructions   A1-8, A3-26
    extension space   A3-40

  Thumb   A6-15
  Thumb   A6-15, A6-17
    examples   A6-16
Load and Store Coprocessor
  addressing modes   A5-60–A5-66
  instructions   A3-33, A3-34
Load and Store Halfword and Load
    Signed Byte
  instructions   A3-26, A3-28
Load and Store Multiple
  addressing modes   A5-52–A5-59
    post-decrement   A1-9
    post-increment   A1-9
    pre-decrement   A1-9
    pre-increment   A1-9
  instructions   A1-9, A3-29, A3-30
    base address   A1-9
  Thumb instructions   A6-18, A6-19
Load and Store Word and Unsigned
    Byte
  addressing modes   A5-19
Load Coprocessor (LDC2) instruction
  A4-33
Load Coprocessor (LDC) instruction
  A4-33
Load instructions   A3-26
  addressing modes   A3-25
  ARM   A3-25
  floating-point values   C1-4, C2-17
Load Multiple (LDMIA) instruction
  Thumb   A7-45
Load Multiple (LDM) instruction
  ARM   A4-37
Load Register Byte with Translation
    (LDRBT) instruction   A4-47
Load Register Byte (LDRB)
    instruction
  ARM   A4-45
  Thumb
    register offset   A7-55
Load Register Doubleword (LDRD)
    instruction   A4-49, A4-51,
    A4-208
Load Register Halfword (LDRH)
    instruction
  ARM   A4-54
Load Register Signed Byte (LDRSB)
    instruction
  ARM   A4-55

  Thumb   A7-60
Load Register Signed Halfword
    (LDRSH) instruction
  ARM   A4-57
  Thumb   A7-61
Load Register with Translation
    (LDRT) instruction   A4-59
Load Register (LDR) instruction
  ARM   A4-42
  Thumb
    PC-relative   A7-50
Load-Store architecture   A1-2
Logical OR (ORR) instruction
  ARM   A4-83
  Thumb   A7-80
Logical Shift Left (LSL) instruction
  ARM   A5-10
    as addressing mode   A5-23,
    A5-29, A5-35
    immediate   A5-9
  Thumb
    register   A7-65
Logical Shift Right (LSR) instruction
  ARM   A5-11, A5-12
    as addressing mode   A5-23,
    A5-29, A5-35
  Thumb
    immediate   A7-67
    register   A7-69
Long branch   A3-9
LSL instruction
  ARM   A5-9, A5-10, A5-23,
    A5-29, A5-35
  Thumb   A7-65
LSR instruction
  ARM   A5-11, A5-12, A5-23,
    A5-29, A5-35
  Thumb   A7-67, A7-69

M

MCR instruction   A4-61
MCRR instruction   A4-63
MCR2 instruction   A4-61
Memory
  aborts   A1-3
  protection   A1-3
  virtual   A1-3

ARM DDI 0100F-02

ARM DDI 0100F-02

Copyright © 1996–2003 ARM Limited. All rights reserved.
Confidential - Draft

W

Z

Numerics

ARM DDI 0100F-02