ARM Architecture Reference Manual

Copyright © 1996–2000 ARM Limited. All rights reserved.

Release Information

The following changes have been made to this document.

<div align="right">**Change History**</div>

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.

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.

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 iv
- *Architecture versions and variants* on page v
- *Using this manual* on page x
- *Conventions* on page xii.

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. In order 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 in order 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 extends 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:

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

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.

- 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.

       ARM DDI 0100E

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 viii.

Long multiply instructions (M variants)

M variants of the ARM instruction set include four extra instructions which perform $32 \times 32 \to 64$ multiplications and $32 \times 32 + 64 \to 64$ 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 A10 *Enhanced DSP Extension*, and include:

• Several new 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.

In order 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* on page viii. 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 ix 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 v.

 ARM DDI 0100E

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

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 quick 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.

Chapter A8 Gives information on the 26-bit architectures (ARMv1, ARMv2, and ARMv2a), and about the backwards-compatibility support for these architectures that is built into some later ARM processors. All of these features are now obsolete, and information about them is only relevant to historical systems.

Chapter A9 Contains some examples of using the ARM instruction set.

Chapter A10 Gives a description of the extra instructions added in the enhanced DSP extension (see *Enhanced DSP instructions (E variants)* on page vii).

 ARM DDI 0100E

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 B2   Gives a general description of the System Control coprocessor and its use.

Chapter B3   Describes the standard ARM memory and system architecture based on the use of a *Memory Management Unit* (MMU). (Chapter B2 and Chapter B5 are also relevant to this architecture.)

Chapter B4   Gives a description of the simpler standard ARM memory and system architecture based on the use of a Protection Unit. (Chapter B2 and Chapter B5 are also relevant to this architecture.)

Chapter B5   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.

Chapter B6   Describes the *Fast Context Switch Extension* (FCSE), which allows switching between multiple small processes ($\leq$ 32MB in size) without incurring large performance costs due to cache flushing and similar overheads.

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.

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 A48 uses ARM addressing mode 1 (see *Addressing Mode 1 - Data-processing operands* on page A52). 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.

 ARM DDI 0100E

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.

Contents
ARM Architecture Reference Manual

Part A: CPU Architecture

Part B: Memory and System Architectures

Part C: Vector Floating-point Architecture

ARM DDI 0100E

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 A1-5.

Copyright © 1996-2000 ARM Limited. All rights reserved.

1.1 About the ARM architecture

The ARM architecture has been designed to allow very small, yet high-performance implementations. The architectural simplicity of ARM processors leads to very small implementations, and small implementations allow devices with very low power consumption.

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 gives you:

• 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 all instructions to maximize execution throughput.

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

1.1.1 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) 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. 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.

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, please refer to *Registers* on page A2-4.

1.1.2   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.

For more details on exceptions, please refer to *Exceptions* on page A2-13.

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.

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.

1.1.3 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)
* 2 interrupt disable bits, one for each type of interrupt
* 5 bits which encode the current processor mode
* 1 bit which encodes whether ARM or Thumb instructions are being executed.

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, please refer to *Program status registers* on page A2-9.

         ARM DDI 0100E

1.2   ARM instruction set

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

*   *Branch instructions*
*   *Data-processing instructions* on page A1-6
*   *Status register transfer instructions* on page A1-7
*   *Load and store instructions* on page A1-7
*   *Coprocessor instructions* on page A1-8
*   *Exception-generating instructions* on page A1-9.

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 is used for a few instructions which do not allow conditional execution.

1.2.1   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 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 instruction set. These allow 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*.

1.2.2 Data-processing instructions

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

- *Arithmetic/logic instructions*
- *Comparison instructions*
- *Multiply instructions*
- *Count Leading Zeros instruction* on page A1-7.

Arithmetic/logic instructions

There are twelve arithmetic/logic instructions which 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

There are four comparison instructions which 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.

Multiply instructions

Multiply instructions come in two classes. Both types multiply two 32-bit register values and store their result:

32-bit result   Normal. Stores the 32-bit result in a register.

64-bit result   Long. Stores the 64-bit result in two separate registers.

Both types of multiply instruction can optionally perform an accumulate operation.

Count Leading Zeros instruction

The Count Leading Zeros (CLZ) instruction determines the number of zero bits at the most significant end of a register value, up to the first 1 bit. This number is written to the destination register of the CLZ instruction.

1.2.3 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.

1.2.4 Load and store instructions

The following load and store instructions are available:

- *Load and Store Register*
- *Load and Store Multiple registers* on page A1-8
- *Swap register and memory contents* on page A1-8.

Load and Store Register

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

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

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.

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.

1.2.5   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.

1.2.6 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.

Chapter A2
Programmer's Model

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

2.1 Data types

ARM processors support the following data types:

Byte   8 bits.

Halfword   16 bits (halfwords must be aligned to two-byte boundaries).

Word   32 bits (words must be aligned to four-byte boundaries).

——— **Note** ———

• All three types are supported in ARM architecture version 4 and above. Only bytes and words were supported prior to ARM architecture version 4.

• 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.

• All data operations, for example ADD, are performed on word quantities.

• Load and store operations can transfer bytes, halfwords and words 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).

 ARM DDI 0100E

2.2 Processor modes

The ARM architecture supports the seven processor modes shown in Table 2-1.

Table 2-1 ARM version 4 processor modes

Processor mode		Description
User	usr	Normal program execution mode
FIQ	fiq	Supports a high-speed data transfer or channel process
IRQ	irq	Used for general-purpose interrupt handling
Supervisor	svc	A protected mode for the operating system
Abort	abt	Implements virtual memory and/or memory protection
Undefined	und	Supports software emulation of hardware coprocessors
System	sys	Runs privileged operating system tasks (ARM architecture version 4 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. While 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 A2-13). 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 which 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.

2.3   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 A2-5.

*   6 status registers. These registers are also 32 bits wide, but only 12 of the 32 bits are allocated or need to be implemented. These are described in *Program status registers* on page A2-9.

Registers are arranged in partially overlapping banks, with a different register bank for each processor mode, as shown in Figure 2-1. 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 2-1 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 2-1 Register organization

Copyright © 1996-2000 ARM Limited. All rights reserved.

2.4 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 A2-7.

2.4.1 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.

2.4.2 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. Examples of this usage can be found in *Single-channel DMA transfer* on page A9-13 and *Dual-channel DMA transfer* on page A9-13.

Registers R13 and R14 have six banked physical registers each. One is used in User and System modes, while 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. In the ARM instruction set, this is by convention only, as there are no defined instructions or other functionality which use R13 in a special-case manner. However, 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, which 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 A2-13 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.)

         ARM DDI 0100E

2.4.3 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 A2-8.

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 3 and below, bits[1:0] of a value written to R15 are ignored, so that the actual destination address of the instruction is (value written to R15) AND 0xFFFFFFFC.

• 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.

 ARM DDI 0100E

2.5 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 which 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		8	7	6	5	4	3	2	1	0
N	Z	C	V	Q		DNM(RAZ)		I	F	T	M4	M3	M2	M1	M0
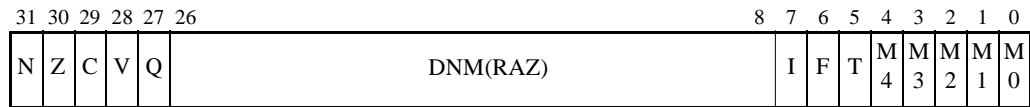
2.5.1 The condition code flags

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 (which 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 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. For more details of the Q flag, see Chapter A10 *Enhanced DSP Extension*.

In architecture versions prior to version 5, and in non-E variants of architecture version 5 and above, bit[27] of the CPSR and SPSRs should be treated as described in *Other bits* on page A2-12.

2.5.2   The control bits

The bottom eight bits of a *Program Status Register* (PSR), incorporating 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

I and F are the interrupt disable bits:

I bit   Disables IRQ interrupts when it is set.

F bit   Disables FIQ interrupts when it is set.

                       ARM DDI 0100E

The T bit

The T bit *should be zero* (SBZ) on ARM architecture version 3 and below, and 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.

On non-T variants of ARM architecture version 5 and above, 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 A2-15).

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, and these determine the mode in which the processor operates. Their interpretation is shown in Table 2-2.

Table 2-2 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 (ARM architecture v4 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. See also Table 8-1 on page A8-9 for details of the mode bits in the 26-bit architectures.

2.5.3 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 which has unexpected side-effects on future versions of the architecture. See the usage notes for the MSR instruction on page A4-62 for more details.

2.6 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 2-3 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*.

―――― **Note** ――――

The normal vector at address `0x00000014` and the high vector at address `0xFFFF0014` are not normally used and are reserved for future expansion. The reserved vector at address `0x00000014` was used for an Address Exception vector in earlier versions of the ARM architecture which had a 26-bit address space. See Chapter A8 *The 26-bit Architectures* for more information.

Table 2-3 Exception processing modes

Exception type	Mode	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`
IRQ (interrupt)	IRQ	`0x00000018`	`0xFFFF0018`
FIQ (fast interrupt)	FIQ	`0x0000001C`	`0xFFFF001C`

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 */
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 A4-34.

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}^
```

2.6.1 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 */
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. See *About the MMU architecture* on page B3-2 for more information on the effects of Reset.

——— **Note** ———

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

2.6.2 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 A3-27).

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 mode */
CPSR[5]  = 0                   /* Execute in ARM state */
/* CPSR[6] is unchanged */
CPSR[7]  = 1                   /* Disable normal interrupts */
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
```

2.6.3 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 */
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
```

2.6.4 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 ARM architecture version 5 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 */
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.

2.6.5 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 */
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, which can include the PC.

- By specifying *base register writeback*, 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 writeback 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 writeback 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 writeback is specified, the value left in the base register is determined by the *abort model* of the implementation, as described in *Abort models*.

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 which 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 which specifies base register writeback, the value in the base register is unchanged.

Base Updated Abort Model

If a Data Abort occurs in an instruction which specifies base register writeback, the base register writeback still occurs.

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.

——— **Note** ———

In some ARMv3 and earlier implementations, a third abort model (the *Early Abort Model*) was used. In this model, base register writeback occurred for LDC, LDM, STC and STM instructions, and the base register was unchanged for all other instructions.

The Early Abort Model is not valid in ARM architecture versions 3M, 4 and above.

Some of these implementations optionally allowed a *Late Abort Model* to be selected. This is identical to the Base Updated Abort Model.

2.6.6 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 2-4 on page A2-20), 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 */
if high vectors configured then
    PC    = 0xFFFF0018
else
    PC    = 0x00000018
```

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.

2.6.7 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 */
if high vectors configured then
    PC    = 0xFFFF001C
else
    PC    = 0x0000001C
```

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.

2.6.8 Exception priorities

Table 2-4 shows the exception priorities:

Table 2-4 Exception priorities

Priority		Exception
Highest	1	Reset
	2	Data Abort
	3	FIQ
	4	IRQ
	5	Prefetch Abort
Lowest	6	Undefined instruction SWI

Undefined instruction and software interrupt cannot occur at the same time, as they each correspond to particular (non-overlapping) decodings of the current instruction, and both must be lower priority than prefetch abort, as 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).

2.6.9 High vectors

Some ARM implementations 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*.

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.

The ARM instruction set does not contain any instructions which can directly change whether normal or high vectors are configured. However, if the standard System Control coprocessor is attached to an ARM processor which 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 register* on page B2-13).

2.7 Memory and memory-mapped I/O

This section discusses memory and memory-mapped I/O, mainly with regard to the assumptions ARM processor implementations make about how their memory systems behave and how programs should access memory. As a result, it describes the memory system *from the outside*, specifying how it should behave without going into much detail of how this behavior can or should be implemented. More details of how some standard memory systems behave can be found in *Part B: Memory and System Architectures*.

The ARM architecture allows a wide variety of memory system designs, using the range of memory and I/O devices which are available. This makes it difficult to specify absolute rules about how a memory system should behave.

Many of the rules below can be broken if the hardware and software are designed appropriately. However, breaking these rules is discouraged, for the following reasons:

- It might make implementing the memory system more difficult.

- It might cause difficulties in porting the system (hardware and/or software) to future ARM processors.

- Standard software (such as compilers and other software toolkit components) might not work with the rule-breaking system.

2.7.1 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.

——— **Note** ———

Prior to ARM architecture version 3, the address space was only 2^{26} bytes, with addresses running from 0 to 2^{26} - 1. This address space was split into 2^{24} words.

Some implementations of subsequent non-T variants of the ARM architecture include backwards-compatibility features to allow execution of code written for this address space. These features are described in Chapter A8 *The 26-bit Architectures*. Their use for any purpose other than executing old code is strongly discouraged.

These backwards-compatibility features are not compatible with T variants of the architecture, due to conflicting uses of bits[1:0] of R15.

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.

—— **Note** ——

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.

2.7.2   Endianness

The rules in *Address space* on page A2-22 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, Table 2-5 and Table 2-6 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 2-5 Big-endian memory system

31 24	23 16	15 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 2-6 Little-endian memory system

31 20	19 12	11 10 9 8 5	4 3 2 1 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

It is IMPLEMENTATION DEFINED whether an ARM implementation supports little-endian memory systems, big-endian memory systems, or both.

The ARM instruction set does not contain any instructions that directly select the endianness. Instead, an ARM implementation which supports both endiannesses has a hardware input to configure it to match the endianness of the attached memory system. If a standard System Control coprocessor is attached to such an ARM implementation, this configuration input can be changed by writing to bit[7] of register 1 of the System Control coprocessor (see *Register 1: Control register* on page B2-13).

If an ARM implementation is configured for a memory system of one endianness but is actually attached to a memory system of the opposite endianness, only word-sized instruction fetches, data loads and data stores can be relied upon. Other memory accesses have UNPREDICTABLE results.

When the standard System Control coprocessor is attached to an ARM processor that supports both endiannesses, bit[7] of the coprocessor's register 1 is cleared on reset. This means that the ARM processor is configured for a little-endian memory system immediately after reset. If 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 of a Thumb instruction being executed.

───── **Note** ─────

The rules on endianness imply that word loads and stores are not affected by the configured endianness. Because of this, it is not possible to reverse the order of the bytes in a word by storing it, changing the configured endianness, and reloading the stored word. Instead, use one of the code sequences in *Swapping endianness* on page A9-4.

More generally, there is no point in changing the configured endianness of an ARM processor to be different from that of the memory system it is attached to, because no additional architecturally defined operations become available as a result of doing so. So normally, the only time the configured endianness is changed is at reset to make it match the memory system endianness.

2.7.3   Unaligned memory accesses

The ARM architecture normally expects all memory accesses to be suitably aligned. In particular, the address used for a word access should normally be word-aligned, and the address used for a halfword access should normally be halfword-aligned. Memory accesses which are not aligned in this way are called *unaligned* memory accesses.

Unaligned instruction fetches

If an address which is not word-aligned is written to R15 during ARM state execution, the result is normally either UNPREDICTABLE or that bits[1:0] of the address are ignored. If an address which is not halfword-aligned is written to R15 during Thumb state execution, bit[0] of the address is normally ignored. See *Writing the program counter* on page A2-8 and individual instruction descriptions for more details. As a result, a value which is read from R15 during execution of valid code always has bits[1:0] zero for ARM state execution and bit[0] zero for Thumb state execution.

When it is specified that these bits are ignored, ARM implementations are not required to ensure that they are cleared from the address sent to memory for the instruction fetch. They can instead send the value written to R15 unchanged to memory, and require the memory system to ignore bits[1:0] of the address for an ARM instruction fetch and bit[0] for a Thumb instruction fetch.

Unaligned data accesses

The architecturally defined behavior of a load/store instruction which generates an unaligned access is one of the following:

- It is UNPREDICTABLE.

- It ignores the low-order address bits that make the access unaligned. This means it effectively uses the formula (address AND 0xFFFFFFFE) for a halfword access, and uses the formula (address AND 0xFFFFFFFC) for a word access.

- It ignores the low-order address bits that make the access unaligned for the memory access itself, but then uses those low-order bits to control a rotation of the loaded data. (This behavior applies only to the LDR and SWP instructions.)

Which of these three options applies to a load/store instruction depends on which instruction it is, and is documented on the individual instruction pages.

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.

——— **Note** ———

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 the *Glossary*.

2.7.4   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, any instructions that have already been prefetched from sequential locations beyond the branch 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.

——— **Note** ———

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 (see *Instruction cache coherency* on page B5-11). 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. Programming techniques that involve the use of self-modifying code are to be avoided as far as possible.

Instruction Memory Barriers (IMBs)

In many systems, however, it is impossible to avoid the use of self-modifying code entirely. For example, any system which allows a program to be loaded into memory and then executed is using self-modifying code.

Each implementation therefore defines 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 often depends 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.

Because the exact sequence of operations to be performed by an IMB depends on the ARM and memory system implementations, it is recommended that software is designed so that the IMB sequence is provided as a call to an easily replaceable *system dependencies* module, rather than being included in-line where it is needed. This eases porting to other ARM processors and memory systems.

Also, in many implementations, the IMB sequence includes operations that are only usable from privileged processor modes, such as the cache cleaning and invalidation operations supplied by the standard System Control coprocessor (see Chapter B5 *Caches and Write Buffers*). To allow User mode programs to use the IMB sequence, it is recommended that it is supplied as an operating system call, invoked by a `SWI` instruction.

In systems that use the 24-bit immediate in a `SWI` instruction to specify the required operating system service, it is recommended that the IMB sequence is requested by the instruction:

```
SWI 0xF00000
```

This call takes no parameters and 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.

Some implementations can use knowledge of the range of addresses to which new instructions have been stored to reduce the execution time cost of an IMB. It is therefore also recommended that a second operating system call is supplied which does an IMB with respect to a specified address range only. On systems that use the 24-bit immediate in a `SWI` instruction to specify the required operating system service, this should be requested by the instruction:

```
SWI 0xF00001
```

and 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).

——— **Note** ———

* When the standard ARM Procedure Calling 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 *LDM (3)* on page A4-34.

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:

- *B, BL* on page A4-10
- *BLX (1)* on page A4-16
- *B (1)* on page A7-18
- *B (2)* on page A7-20
- *BL, BLX(1)* on page A7-26.

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 occasionally referred to elsewhere in this manual as the set of instructions that can be reached by *predictable subsequent execution* from the MSR instruction.

2.7.5 Memory-mapped I/O

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 in behavior mainly affect the use of caches and write buffers in the memory system. This is discussed in Chapter B5 *Caches and Write Buffers*. In short, memory-mapped I/O locations are normally marked as uncachable and unbufferable, to avoid changes to the number, type, order, or timing of the accesses made to them.

Instruction fetches from memory-mapped I/O

As described in *Prefetching and self-modifying code* on page A2-27, 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. Any system design which relies on executing instructions fetched from a memory-mapped I/O location is likely to be hard to port to future ARM implementations.

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 IMPLEMENTATION DEFINED 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.

Typical requirements include:

- Constraints on memory attributes of the memory-mapped I/O locations. For example, in the standard memory system architectures described in *Part B: Memory and System Architectures*, the memory locations must be uncachable and unbufferable.

- 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 described in *Part B: Memory and System Architectures*, the time order of the memory accesses is not guaranteed to be preserved if the `LDM` or `STM` crosses the boundary between a cachable area of memory and an uncachable, unbufferable area. Such `LDM` and `STM` instructions are therefore not suitable for memory-mapped I/O.

Chapter A3
The ARM Instruction Set

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

3.1 Instruction set encoding

Figure 3-1 shows the ARM instruction set encoding.

All other bit patterns are UNPREDICTABLE or UNDEFINED. See *Extending the instruction set* on page A3-27 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 3-3	cond [1]	0 0 0	1 0 x x	0	x x x x	x x x x	x x x x	x	x x	0	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 3-3	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	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
Undefined instruction	cond [1]	0 1 1	x x x x		x x x x	x x x x	x x x x	x	x x	1	x x x x
Undefined instruction [4,7]	1 1 1 1	0 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
Load/store multiple	cond [1]	1 0 0	P U S W	L	Rn	register list					
Undefined instruction [4]	1 1 1 1	1 0 0	x x x x		x x x x	x x x x	x x x x	x	x x	x	x x x x
Branch and branch with link	cond [1]	1 0 1	L		24-bit offset						
Branch and branch with link and change to Thumb [4]	1 1 1 1	1 0 1	H		24-bit offset						
Coprocessor load/store and double register transfers [6]	cond [5]	1 1 0	P U N W	L	Rn	CRd	cp_num	8-bit offset			
Coprocessor data processing	cond [5]	1 1 1 0	opcode1		CRn	CRd	cp_num	opcode2		0	CRm
Coprocessor register transfers	cond [5]	1 1 1 0	opcode1	L	CRn	Rd	cp_num	opcode2		1	CRm
Software interrupt	cond [1]	1 1 1 1			swi number						
Undefined instruction [4]	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 3-1 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 ARM architecture version 4.
4. UNPREDICTABLE prior to ARM architecture version 5.
5. If the `cond` field is 1111, this instruction is UNPREDICTABLE prior to ARM architecture version 5.
6. The coprocessor double register transfer instructions are described in Chapter A10 *Enhanced DSP Extension*.
7. In E variants of architecture version 5 and above, the cache preload instruction `PLD` uses a small number of these instruction encodings.

3.1.1 Multiplies and extra load/store instructions

Figure 3-2 shows extra multiply and 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 18 17 16	15 14 13 12	11 10 9 8	7	6	5	4	3 2 1 0
Multiply (accumulate)	cond	0 0 0 0	0	0	A	S	Rd	Rn	Rs	1	0	0	1	Rm
Multiply (accumulate) long	cond	0 0 0 0	1	U	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 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/store two words register offset [2]	cond	0 0 0 P	U	0	W	0	Rn	Rd	SBZ	1	1	S	1	Rm
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 two words immediate offset [2]	cond	0 0 0 P	U	1	W	0	Rn	Rd	HiOffset	1	1	S	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

Figure 3-2 Multiplies and extra load/store instructions

1. UNPREDICTABLE prior to ARM architecture version 4.
2. These instructions are described in Chapter A10 *Enhanced DSP Extension*.

——— **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 3-2 or its notes, is an undefined instruction (or UNPREDICTABLE prior to ARM architecture version 4).

3.1.2 Miscellaneous instructions

Figure 3-3 shows 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 [1]	cond	0 0 0 1 0	0	1	0	SBO	SBO	SBO	0 0 0 1	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 [2]	cond	0 0 0 1 0	0	1	0	SBO	SBO	SBO	0 0 1 1	Rm
Enhanced DSP add/subtracts [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
Enhanced DSP multiplies[4]	cond	0 0 0 1 0	op		0	Rd	Rn	Rs	1 y x 0	Rm

Figure 3-3 Miscellaneous instructions

1. Defined in ARM architecture version 5 and above, and in T variants of ARM architecture version 4.
2. This is an undefined instruction is ARM architecture version 4, and is UNPREDICTABLE prior to ARM architecture version 4.
3. If the cond field of this instruction is not 1110, it is UNPREDICTABLE.
4. The enhanced DSP instructions are described in Chapter A10 *Enhanced DSP Extension*.

——— **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 3-3 or its notes, is an undefined instruction (or UNPREDICTABLE prior to architecture version 4).

3.2 The condition field

Almost all 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.

Every instruction contains a 4-bit condition code field in bits 31 to 28:

```
31      28 27                                                        0
┌─────────┬─────────────────────────────────────────────────────────┐
│  cond   │                                                         │
└─────────┴─────────────────────────────────────────────────────────┘
```
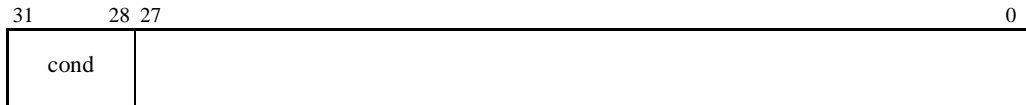
This field contains one of the 16 values described in Table 3-1 on page A3-6. Most instruction mnemonics can be extended with the letters defined in the mnemonic extension field.

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.

3.2.1 Condition code 0b1111

As indicated in Table 3-1 on page A3-6, 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. All instruction encoding diagrams which show bits[31:28] as `cond` only match instructions in which these bits are not equal to 0b1111, unless otherwise stated in the individual instruction description.

Table 3-1 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	(NV)	See *Condition code 0b1111* on page A3-5	-

3.3   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 trap. See *The control bits* on page A2-10 for more details.

3.3.1   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
```

3.3.2   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.

3.4 Data-processing instructions

ARM has 16 data-processing instructions, shown in Table 3-2.

Table 3-2 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 A3-27). 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 A2-9 for more details.

3.4.1   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
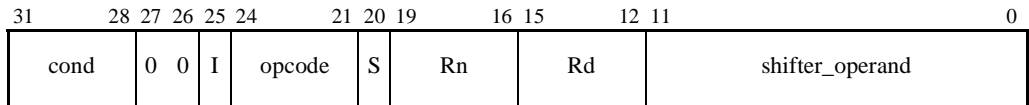
I bit   Distinguishes between the immediate and register forms of
`<shifter_operand>`.

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.

3.4.2   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 A4-23.

CMP   Compare. See *CMP* on page A4-25.

EOR   Logical EOR. See *EOR* on page A4-26.

MOV   Move. See *MOV* on page A4-56.

MVN   Move Negative. See *MVN* on page A4-68.

ORR   Logical OR. See *ORR* on page A4-70.

RSB   Reverse Subtract. See *RSB* on page A4-72.

RSC   Reverse Subtract with Carry. See *RSC* on page A4-74.

SBC   Subtract with Carry. See *SBC* on page A4-76.

SUB   Subtract. See *SUB* on page A4-98.

TEQ   Test Equivalence. See *TEQ* on page A4-106.

TST   Test. See *TST* on page A4-107.

3.5　Multiply instructions

ARM has two classes of Multiply instruction:

- normal, 32-bit result
- long, 64-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 due to the efficiency of shift and add, or shift and reverse subtract instructions.

3.5.1　Normal multiply

There are two Multiply instructions that produce 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 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.

3.5.2　Long multiply

There are four Multiply instructions that produce 64-bit results (long multiply).

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.

The remaining 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.

All four long multiply instructions can optionally set the N (Negative) and Z (Zero) condition code flags.

3.5.3　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
```

3.5.4   List of multiply instructions

MLA   Multiply Accumulate. See *MLA* on page A4-54.

MUL   Multiply. See *MUL* on page A4-66.

SMLAL   Signed Multiply Accumulate Long. See *SMLAL* on page A4-78.

SMULL   Signed Multiply Long. See *SMULL* on page A4-80.

UMLAL   Unsigned Multiply Accumulate Long. See *UMLAL* on page A4-109.

UMULL   Unsigned Multiply Long. See *UMULL* on page A4-111.

3.6 Miscellaneous arithmetic instructions

In addition to the normal data-processing and multiply instructions, versions 5 and above of the ARM architecture 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 zero). Two typical applications for this are:

* To determine how many bits the operand should be shifted left in order 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.

3.6.1 Instruction encoding

CLZ{<cond>}   <Rd>, <Rm>

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

Rd   Specifies the destination register.

Rm   Specifies the operand register.

3.6.2 List of miscellaneous arithmetic instructions

CLZ   Count Leading Zeros. See *CLZ* on page A4-22.

3.7 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.

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 A10-5 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 A2-11).

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.

3.7.1 CPSR value

Altering the value of the CPSR has three 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).

——— **Note** ———

The T bit must not be changed directly by writing to the CPSR, but only via the BX instruction, and in the implicit SPSR to CPSR moves in instructions designed for exception return. Attempts to enter or leave Thumb state by directly altering the T bit can have UNPREDICTABLE consequences.

3.7.2   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
```

3.7.3   List of status register access instructions

MRS   Move PSR to General-purpose Register. See *MRS* on page A4-60.

MSR   Move General-purpose Register to PSR. See *MSR* on page A4-62.

                                   ARM DDI 0100E

3.8   Load and store instructions

The ARM architecture supports two broad types of instruction which load or store the value of a single register 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.

3.8.1   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.

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.

3.8.2 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
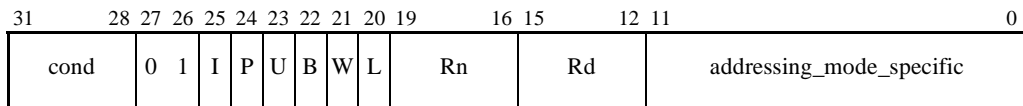
I, P, U, W Are bits that distinguish between different types of <addressing_mode>.

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.

3.8.3 Load and Store Halfword and Load Signed Byte

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 Halfword and Load Signed Byte 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
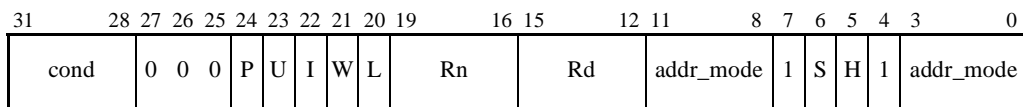
addr_mode Are addressing-mode-specific bits.

I, P, U, W Are bits that specify the type of addressing mode (see *Addressing Mode 3 - Miscellaneous Loads and Stores* on page A5-34).

L bit Distinguishes between a Load (L==1) and a Store instruction (L==0).

S bit Distinguishes between a signed (S==1) and an unsigned (S==0) halfword access. If the L bit is zero and S bit is one, the instruction is UNPREDICTABLE.

 ARM DDI 0100E

H bit   Distinguishes between a halfword (H==1) and a signed byte (H==0) access. If the S bit and H bit are both zero, this instruction encodes a SWP or Multiply instruction.

Rn   Specifies the base register used by the addressing mode.

Rd   Specifies the register whose contents are to be loaded or stored.

3.8.4   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
```

3.8.5   List of load and store instructions

LDR   Load Word. See *LDR* on page A4-37.

LDRB   Load Byte. See *LDRB* on page A4-40.

LDRBT   Load Byte with User Mode Privilege. See *LDRBT* on page A4-42.

LDRH   Load Unsigned Halfword. See *LDRH* on page A4-44.

LDRSB   Load Signed Byte. See *LDRSB* on page A4-46.

LDRSH   Load Signed Halfword. See *LDRSH* on page A4-48.

LDRT   Load Word with User Mode Privilege. See *LDRT* on page A4-50.

STR   Store Word. See *STR* on page A4-88.

STRB   Store Byte. See *STRB* on page A4-90.

STRBT   Store Byte with User Mode Privilege. See *STRBT* on page A4-92.

STRH   Store Halfword. See *STRH* on page A4-94.

STRT   Store Word with User Mode Privilege. See *STRT* on page A4-96.

3.9   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	
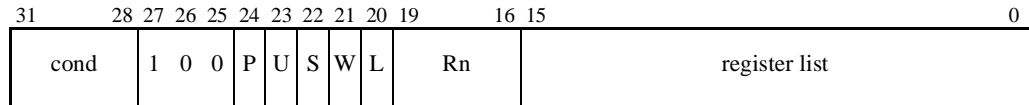
register list   The list of <registers> has one bit for each general-purpose register. Bit 0 is for R0, and bit 15 is for R15 (the PC).

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 A5-48).

S bit   For LDMs 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 STMs, and LDMs 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.

3.9.1   Examples

```
STMFD    R13!, {R0 - R12, LR}
LDMFD    R13!, {R0 - R12, PC}
LDMIA    R0, {R5 - R8}
STMDA    R1!, {R2, R5, R7 - R9, R11}
```

3.9.2 List of Load and Store Multiple instructions

LDM   Load Multiple. See *LDM (1)* on page A4-30.

LDM   User Registers Load Multiple. See *LDM (2)* on page A4-32.

LDM   Load Multiple with Restore CPSR. See *LDM (3)* on page A4-34.

STM   Store Multiple. See *STM(1)* on page A4-84.

STM   User Registers Store Multiple. See *STM (2)* on page A4-86.

Copyright © 1996-2000 ARM Limited. All rights reserved.

3.10   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.

3.10.1   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
```

3.10.2   List of semaphore instructions

SWP   Swap. See *SWP* on page A4-102.

SWPB   Swap Byte. See *SWPB* on page A4-104.

3.11   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 A2-16). 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 A2-16). 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.

3.11.1   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
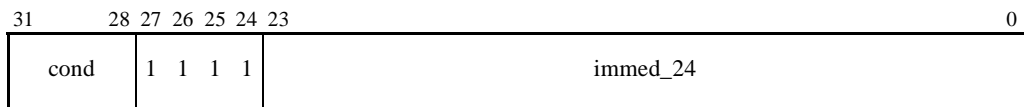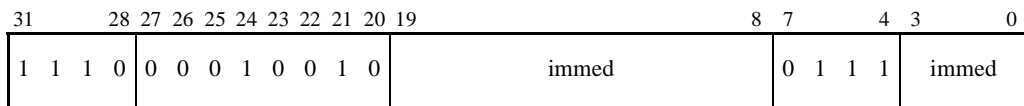
In both SWI and BKPT, the immediate fields of the instruction are ignored by the ARM processor. The SWI or Prefetch Abort handler can optionally be written to load the instruction that caused the exception and extract these fields. This allows them to be used to communicate extra information about the Operating System call or breakpoint to the handler.

3.11.2   List of exception-generating instructions

BKPT   Breakpoint. See *BKPT* on page A4-14.

SWI   Software Interrupt. See *SWI* on page A4-100.

 ARM DDI 0100E

3.12　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 trap, 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.

3.12.1　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
```

3.12.2   List of coprocessor instructions

CDP   Coprocessor Data Operations. See *CDP* on page A4-20.

LDC   Load Coprocessor Register. See *LDC* on page A4-28.

MCR   Move to Coprocessor from ARM Register. See *MCR* on page A4-52.

MRC   Move to ARM Register from Coprocessor. See *MRC* on page A4-58.

STC   Store Coprocessor Register. See *STC* on page A4-82.

——— **Note** ———

Coprocessor instructions are not implemented in ARM architecture version 1.

                                       ARM DDI 0100E

3.13   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 space* on page A3-28
- *Arithmetic instruction extension space* on page A3-29
- *Control instruction extension space* on page A3-30
- *Load/store instruction extension space* on page A3-32
- *Coprocessor instruction extension space* on page A3-33
- *Unconditional instruction extension space* on page A3-34.

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 A3-5 and *Unconditional instruction extension space* on page A3-34 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.

3.13.1   Undefined instruction 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	x	x	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 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	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.

           ARM DDI 0100E

3.13.2   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 3-3 summarizes the instructions that have already been allocated in this area.

Table 3-3 Arithmetic instruction space

Instructions	op1	Architecture versions
MUL, MULS	000S	Version 2 and above
MLA, MLAS	001S	Version 2 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

3.13.3 Control 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 3-4 summarizes the instructions that have already been allocated in this area.

Table 3-4 Control 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
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
QDSUB	0	0101	11	E variants of version 5 and above

Table 3-4 Control extension space instructions (continued)

Instruction	Bit[25]	Bits[7:4]	op1	Architecture versions
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

3.13.4   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 3-5 summarizes the instructions that have already been allocated in this area.

Table 3-5 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
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

   ARM DDI 0100E

3.13.5 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 trap directly.

- The ARM processor might require attached coprocessors not to respond to such instructions. This causes the undefined instruction trap to be taken (see *Undefined Instruction exception* on page A2-15).

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 described in Chapter A10 *Enhanced DSP Extension*.

3.13.6 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 20	19 8	7 4	3 0
1 1 1 1	opcode1	x x x x x x x x x x x x x	opcode2	x x x x
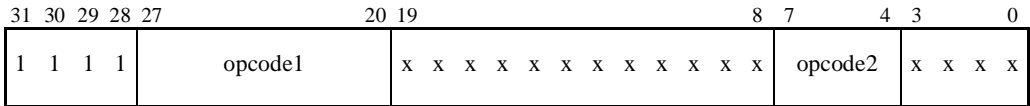
Table 3-6 summarizes the instructions that have already been allocated in this area.

Table 3-6 Unconditional instruction extension space

Instruction	opcode1								opcode2				Architecture versions
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
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 A4-113.

4.1 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.

4.1.1 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 Preface-xiii.

Architecture versions

This gives details of architecture versions where the instruction is valid. For details, see *Architecture versions and variants* on page Preface-v.

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 Preface-xii.

Information on usage

Usage sections are included where appropriate to supply suggestions and other information about how to use the instruction effectively.

4.1.2   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	
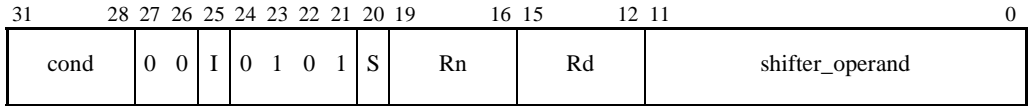
The ADC (Add with Carry) instruction adds the value of <shifter_operand> and the Carry flag to the value of <Rn> and stores the result in <Rd>. The condition code flags are optionally updated, based on the result.

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 A3-5. 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 of the instruction.

<Rn>   Specifies the register that contains the first operand for the addition.

<shifter_operand>

Specifies the second operand for the addition. 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 A3-27 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

ADC is used 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 A5-17 for information on how to perform a similar rotation to the right.

4.1.3 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	
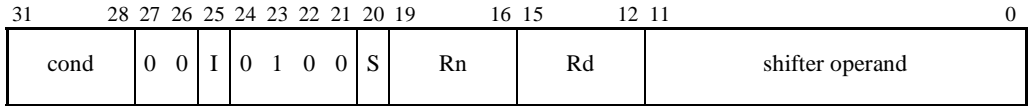
The ADD instruction adds the value of `<shifter_operand>` to the value of register `<Rn>`, and stores the result in the destination register `<Rd>`. The condition code flags are optionally updated, based on the result.

Syntax

ADD{<cond>}{S} <Rd>, <Rn>, <shifter_operand>

where:

<cond>   Is the condition under which the instruction is executed. *The condition field* on page A3-5. 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 of the instruction.

<Rn>   Specifies the register that contains the first operand for the addition.

<shifter_operand>

Specifies the second operand for the addition. 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 A3-27 to determine which instruction it is.

Architecture version

All

       Copyright © 1996-2000 ARM Limited. All rights reserved.   ARM DDI 0100E

Exceptions

None

Operation

```
if ConditionPassed(cond) then
    Rd = Rn + shifter_operand
    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)
        V Flag = OverflowFrom(Rn + shifter_operand)
```

Usage

The ADD instruction is used to add two values together to produce a third.

To increment a register value in Rx use:

```
ADD Rx, Rx, #1
```

Constant multiplication of Rx by 2^n+1 into Rd can be performed 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.

4.1.4 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	
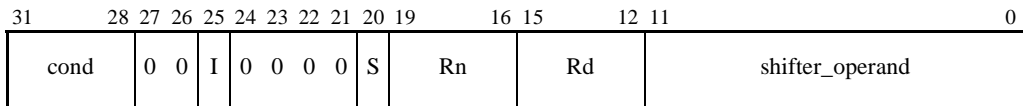
The AND instruction performs a bitwise AND of the value of register <Rn> with the value of
<shifter_operand>, and stores the result in the destination register <Rd>. The condition code flags
are optionally updated, based on the result.

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 A3-5. 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 of the instruction.

<Rn>   Specifies the register that contains the first operand for the operation.

<shifter_operand>

                  Specifies the second operand for the operation. 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 A3-27 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
        CPSR = SPSR
    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.

4.1.5   B, BL

31		28	27	26	25	24	23	0
cond			1	0	1	L	signed_immed_24	
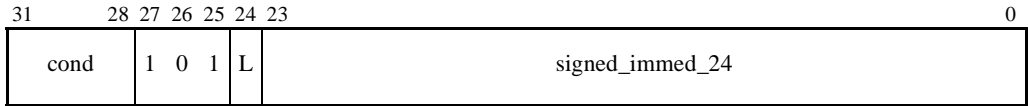
The B (Branch) and BL (Branch and Link) instructions cause a branch to a target address, and provide both conditional and unconditional changes to program flow.

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 A3-5. 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.

Architecture version

All

Exceptions

None

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)
```

Copyright © 1996-2000 ARM Limited. All rights reserved.

Usage

The `BL` instruction is used to perform a subroutine call. The return from subroutine is achieved by copying the LR 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) needs to:

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.

4.1.6   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		
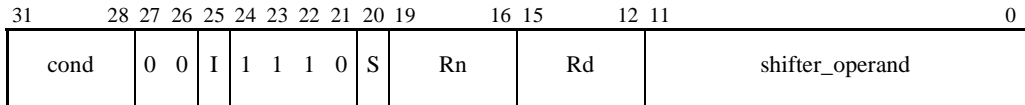
The BIC (Bit Clear) instruction performs a bitwise AND of the value of register <Rn> with the complement of the value of <shifter_operand>, and stores the result in the destination register <Rd>. The condition code flags are optionally updated, based on the result.

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 A3-5. 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 of the instruction.

<Rn>   Specifies the register that contains the first operand for the operation.

<shifter_operand>

                Specifies the second operand for the operation. 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 A3-27 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
        CPSR = SPSR
    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

BIC can be used to clear selected bits in a register. For each bit, BIC with 1 clears the bit, and BIC with 0 leaves it unchanged.

4.1.7 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

The BKPT (Breakpoint) instruction 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   <immediate>
```

where:

<immediate>   Is a 16-bit immediate value, the top 12 bits of which are placed in bits[19:8] of the instruction, and the bottom 4 bits of which 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 the BKPT instruction 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 The BKPT instruction must be 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 will occur.

For more information, consult the documentation for the debug system being used.

4.1.8   BLX (1)

31 30 29 28	27 26 25 24	23	0
1 1 1 1	1 0 1	H	signed_immed_24
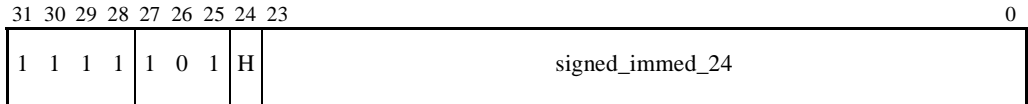
This form of the BLX (Branch with Link and Exchange) instruction is used to call a Thumb subroutine from the ARM instruction set at an address specified in the instruction. This instruction is unconditional (always causing a change in program flow) and preserves the address of the instruction following 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.

Architecture version

Version 5 and above

Exceptions

None

Operation

```
LR = address of the instruction after the BLX instruction
T Flag = 1
PC = PC + (SignExtend(signed_immed_24) << 2) + (H << 1)
```

               ARM DDI 0100E

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) needs to:

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.

4.1.9   BLX (2)
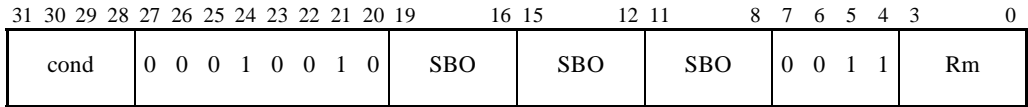
31 30 29 28	27 26 25 24 23 22 21 20	19 16	15 12	11 8	7 6 5 4	3 0
cond	0 0 0 1 0 0 1 0	SBO	SBO	SBO	0 0 1 1	Rm

This form of `BLX` is used to call an ARM or Thumb subroutine from the ARM instruction set, at an address specified in a register. The branch target address is the value of register Rm, with its bit[0] forced to zero. The instruction set to be used at the branch target is chosen by setting the CPSR T bit to bit[0] of Rm.

Register 14 is set to a return address. To return from the subroutine, use a `BX R14` instruction, or store R14 on the stack and re-load 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 A3-5. 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

Exceptions

None

Operation

```
if ConditionPassed(cond) then
    LR = address of instruction after the BLX instruction
    T Flag = 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.

4.1.10 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	

The BX (Branch and Exchange) instruction branches to an address held in a register Rm, with an optional switch to Thumb execution. The branch target address is the value of register Rm, with its bit[0] forced to zero. The instruction set to be used at the branch target is chosen by setting the CPSR T bit to bit[0] of Rm.

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 A3-5. 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, plus T variants of version 4

Exceptions

None

Operation

```
if ConditionPassed(cond) then
    T Flag = 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.

4.1.11 CDP

cond	1 1 1 0	opcode_1	CRn	CRd	cp_num	opcode_2	0	CRm

31   28 27 26 25 24 23   20 19   16 15   12 11   8 7   5 4 3   0

The CDP (Coprocessor Data Processing) instruction 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 A3-5. 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 for the instruction.
<CRm>	Specifies the coprocessor register that contains the second operand for the instruction.
<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

               ARM DDI 0100E

Operation

```
if ConditionPassed(cond) then
    Coprocessor[cp_num]-dependent operation
```

Usage

CDP is used 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 trap.

Copyright © 1996-2000 ARM Limited. All rights reserved.

4.1.12 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	

The CLZ (Count Leading Zeros) instruction returns the number of binary zero bits before the first binary one bit in a register value. The source register is scanned from the most significant bit (bit[31]) towards the least significant bit (bit[0]). The result value is 32 if no bits are set in the source register, and zero if bit[31] is set.

This instruction 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 A3-5. 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

To normalize the value of register Rm, use CLZ followed by a left shift of Rm by the resulting Rd value. 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
```

4.1.13 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	
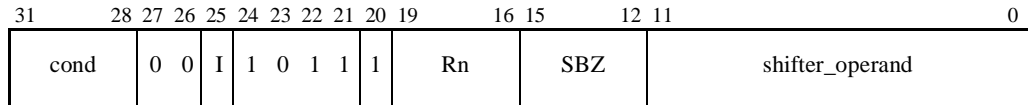
The CMN (Compare Negative) instruction compares a register value with the negative of another arithmetic value. The condition flags are updated, based on the result of adding the second arithmetic value to the register value, so that subsequent instructions can be conditionally executed.

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 A3-5. If <cond> is omitted, the AL (always) condition is used.

<Rn>   Specifies the register that contains the first operand for the operation.

<shifter_operand>

        Specifies the second operand for the operation. 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 A3-27 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

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
```

while this instruction always leaves the C flag = 0:

```
CMN Rn, #0
```
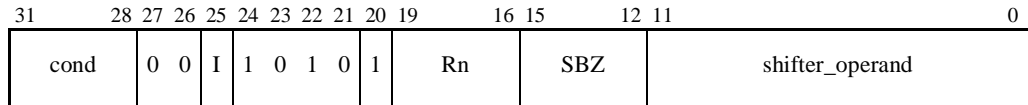
 ARM DDI 0100E

4.1.14 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	

The CMP (Compare) instruction compares a register value with another arithmetic value. The condition flags are updated, based on the result of subtracting the second arithmetic value from the register value, so that subsequent instructions can be conditionally executed.

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 A3-5. If <cond> is omitted, the AL (always) condition is used.

<Rn>   Specifies the register that contains the first operand for the operation.

<shifter_operand>

Specifies the second operand for the operation. 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 A3-27 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)
```

4.1.15 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	
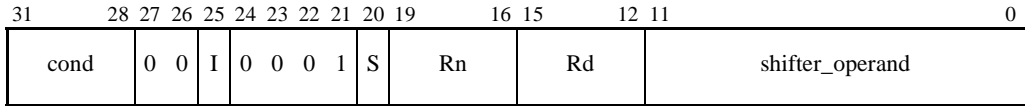
The EOR (Exclusive OR) instruction performs a bitwise Exclusive-OR of the value of register <Rn> with the value of <shifter_operand>, and stores the result in the destination register <Rd>. The condition code flags are optionally updated, based on the result.

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 A3-5. 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 of the instruction.

<Rn>   Specifies the register that contains the first operand for the operation.

<shifter_operand>

                Specifies the second operand for the operation. 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 A3-27 to determine which instruction it is.

Architecture version

All

                   ARM DDI 0100E

Exceptions

None

Operation

```
if ConditionPassed(cond) then
    Rd = Rn EOR shifter_operand
    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 = shifter_carry_out
        V Flag = unaffected
```

Usage

EOR can be used to invert selected bits in a register. For each bit, EOR with 1 inverts that bit, and EOR with 0 leaves it unchanged.

4.1.16   LDC

cond	1 1 0	P	U	N	W	1	Rn	CRd	cp_num	8_bit_word_offset

(bit positions: 31   28 27 26 25 24 23 22 21 20 19   16 15   12 11   8 7   0)
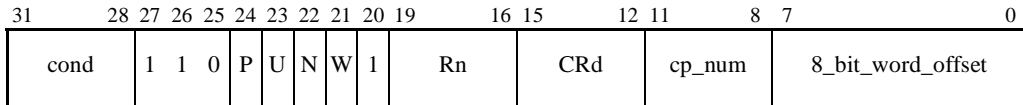
The `LDC` (Load Coprocessor) instruction loads memory data from the sequence of consecutive memory addresses calculated by <addressing_mode> to the coprocessor whose number is cp_num. If no coprocessors indicate that they can execute the instruction, an Undefined Instruction exception is generated.

Syntax

```
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 A3-5. 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 of the instruction.

<addressing_mode>

Is described in *Addressing Mode 5 - Load and Store Coprocessor* on page A5-56. 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 writeback*).

Architecture version

`LDC` is in Version 2 and above.

`LDC2` is in Version 5 and above.

           ARM DDI 0100E

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 A2-17.

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 B2 *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 trap.

4.1.17   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
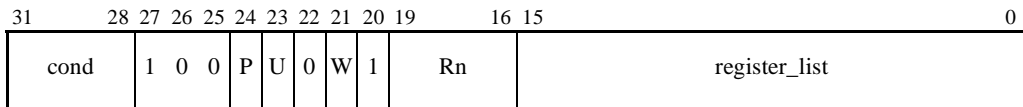
This form of the LDM (Load Multiple) instruction is useful for block loads, stack operations and procedure exit sequences. It loads a non-empty subset, or possibly all, of the general-purpose registers from sequential memory locations.

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. 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 A3-5. If <cond> is omitted, the AL (always) condition is used.

<addressing_mode>

                  Is described in *Addressing Mode 4 - Load and Store Multiple* on page A5-48. 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 A5-48. 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.

                                 ARM DDI 0100E

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 writeback 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 A2-17.

Non word-aligned addresses

Load Multiple instructions ignore the least significant two bits of address (the words are not rotated as for Load Word).

Alignment If an implementation includes a System Control coprocessor (see Chapter B2 *The System Control Coprocessor*), and alignment checking is enabled, an address with bits[1:0] != 0b00 causes an alignment exception.

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 A2-32 for details.

4.1.18 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	

This form of `LDM` loads User mode registers when the processor is in a privileged mode (useful when performing process swaps, and in instruction emulators). The instruction 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 A3-5. If `<cond>` is omitted, the AL (always) condition is used.

`<addressing_mode>`

Is described in *Addressing Mode 4 - Load and Store Multiple* on page A5-48. 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

Banked registers This form of LDM must not be followed by an instruction which accesses banked registers (a following NOP is a good way to ensure this).

Writeback 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 A2-17.

Non word-aligned addresses

LDM instructions ignore the least significant two bits of address (words are not rotated as for Load Word).

Alignment If an implementation includes a System Control coprocessor (see Chapter B2 *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 A2-32 for details.

4.1.19 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	
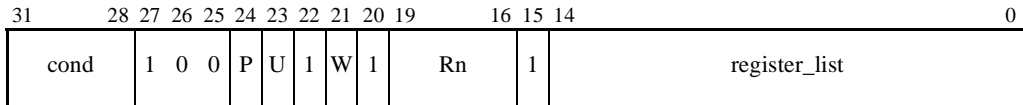
This form of is useful for returning from an exception. It loads a subset (or possibly all) of the general-purpose registers and the PC from sequential memory locations. Also, the SPSR of the current mode is copied to the CPSR.

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. 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 A3-5. If <cond> is omitted, the AL (always) condition is used.

<addressing_mode>

        Is described in *Addressing Mode 4 - Load and Store Multiple* on page A5-48. 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 A5-48). 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

    CPSR = SPSR

    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 writeback 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 A2-17.

Non word-aligned addresses

Load Multiple instructions ignore the least significant two bits of address (the words are not rotated as for Load Word).

Alignment   If an implementation includes a System Control coprocessor (see Chapter B2 *The System Control Coprocessor*), and alignment checking is enabled, an address with bits[1:0] != 0b00 causes an alignment exception.

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 A2-32 for details.

4.1.20   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	
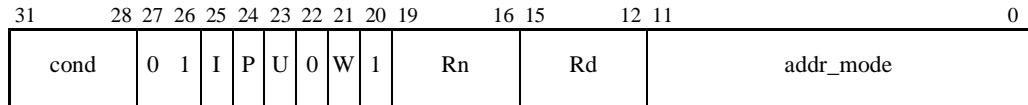
The `LDR` (Load Register) instruction loads a word from the memory address calculated by `<addressing_mode>` and writes it to register `<Rd>`. If the address is not word-aligned, the loaded value is rotated right by 8 times the value of bits[1:0] of the address. For a little-endian memory system, this 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 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. 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 A3-5. 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 A5-18. 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 writeback*).

Architecture version

All

Exceptions

Data Abort

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.

Notes

Data abort   For details of the effects of the instruction if a data abort occurs, see *Effects of data-aborted instructions* on page A2-17.

Operand restrictions

If <addressing_mode> specifies base register writeback, and the same register is specified for <Rd> and <Rn>, the results are UNPREDICTABLE.

Alignment   If an implementation includes a System Control coprocessor (see Chapter B2 *The System Control Coprocessor*), and alignment checking is enabled, an address with bits[1:0] != 0b00 causes an alignment exception.

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, as branches to non word-aligned addresses are not possible in ARM state.

4.1.21   LDRB
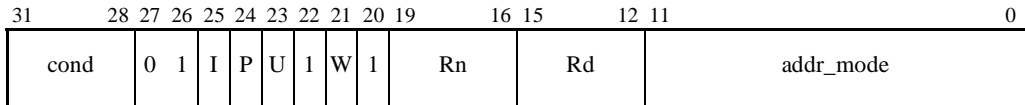
31	28	27	26	25	24	23	22	21	20	19	16	15	12	11	0
cond		0	1	I	P	U	1	W	1	Rn		Rd		addr_mode	

The LDRB (Load Register Byte) instruction loads a byte from the memory address calculated by
<addressing_mode>, zero-extends the byte to a 32-bit word, and writes the word to register <Rd>.

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 A3-5. 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 A5-18.
                 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 writeback*).

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.

 ARM DDI 0100E

Notes

Operand restrictions

If `<addressing_mode>` specifies base register writeback, 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 A2-17.

4.1.22   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	
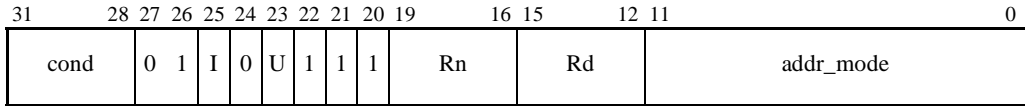
The `LDRBT` (Load Register Byte with Translation) instruction loads a byte from the memory address calculated by `<post_indexed_addressing_mode>`, zero-extends the byte to a 32-bit word, and writes the word to register `<Rd>`.

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

```
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 A3-5. 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 A5-18. 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 writeback*).

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 A2-17.

4.1.23   LDRH
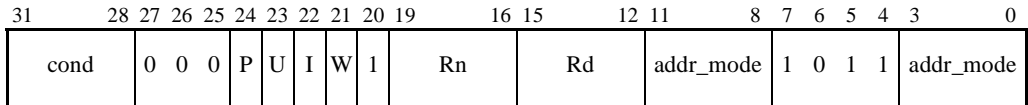
31	28	27	26	25	24	23	22	21	20	19		16	15		12	11		8	7	6	5	4	3		0
cond		0	0	0	P	U	I	W	1	Rn			Rd			addr_mode			1	0	1	1	addr_mode		

The LDRH (Load Register Halfword) instruction loads a halfword from the memory address calculated by <addressing_mode>, zero-extends the halfword to a 32-bit word, and writes the word to register <Rd>. 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 A3-5. 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 A5-34. 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 writeback*).

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 writeback, 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 A2-17.

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 B2 *The System Control Coprocessor*), and alignment checking is enabled, an address with bit[0] != 0 causes an alignment exception.

4.1.24   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	

The LDRSB (Load Register Signed Byte) instruction loads a byte from the memory address calculated by <addressing_mode>, sign-extends the byte to a 32-bit word, and writes the word to register <Rd>.

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 A3-5. 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 A5-34. 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 writeback*).

Architecture version

Version 4 and above

Exceptions

Data Abort

Operation

```
if ConditionPassed(cond) then
    data = Memory[address,1]
    Rd = SignExtend(data)
```

Usage

Used with a suitable addressing mode, LDRSB allows 8-bit signed 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, which facilitates position-independent code.

             ARM DDI 0100E

Notes

Operand restrictions

If `<addressing_mode>` specifies base register writeback, and the same register is specified for `<Rd>` and `<Rn>`, the results are UNPREDICTABLE.
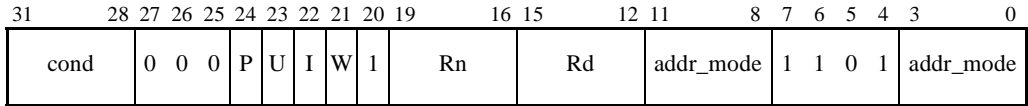
Data abort   For details of the effects of the instruction if a data abort occurs, see *Effects of data-aborted instructions* on page A2-17.

4.1.25 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	

The LDRSH (Load Register Signed Halfword) instruction loads a halfword from the memory address calculated by <addressing_mode>, sign-extends the halfword to a 32-bit word, and writes the word to register <Rd>. 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 A3-5. 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 A5-34. 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 writeback*).

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.
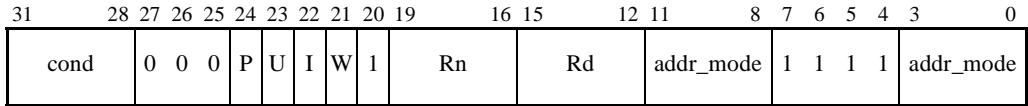
Notes

Operand restrictions

If <addressing_mode> specifies base register writeback, 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 A2-17.

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 B2 *The System Control Coprocessor*), and alignment checking is enabled, an address with bit[0] != 0 causes an alignment exception.

4.1.26   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	

The `LDRT` (Load Register with Translation) instruction loads a word from the memory address calculated by `<addressing_mode>` and writes it to register `<Rd>`. If the address is not word-aligned, the loaded data is rotated as for the `LDR` instruction (see *LDR* on page A4-37).

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

```
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 A3-5. 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 A5-18. 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 writeback*).

Architecture version

All

Exceptions

Data Abort

             ARM DDI 0100E

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 A2-17.

Alignment   If an implementation includes a System Control coprocessor (See Chapter B2 *The System Control Coprocessor*), and alignment checking is enabled, an address with bits[1:0] != 0b00 causes an alignment exception.

4.1.27   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	

The MCR (Move to Coprocessor from ARM Register) instruction 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 A3-5. 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

`MCR` is used to initiate coprocessor instructions that operate on values in ARM registers. 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 trap.

4.1.28 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	
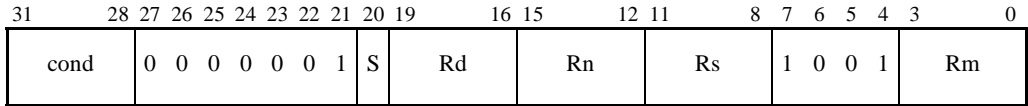
The MLA (Multiply Accumulate) multiplies signed or unsigned operands to produce a 32-bit result, which is then added to a third operand, and written to the destination register. The condition code flags are optionally updated, based on the result.

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 A3-5. 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 of the instruction.

<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    /* See "C flag" note */
        V Flag = unaffected
```

Notes

Use of R15   Specifying R15 for register <Rd>, <Rm>, <Rs>, or <Rn> has UNPREDICTABLE
                        results.

Operand restriction Specifying the same register for <Rd> and <Rm> 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.

4.1.29   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	
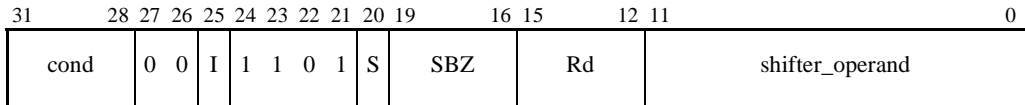
The MOV (Move) instruction moves the value of <shifter_operand> to the destination register <Rd>. The condition code flags are optionally updated, based on the result.

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 A3-5. 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 of the instruction.

<shifter_operand>

              Specifies the operand for the operation. 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 A3-27 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
        CPSR = SPSR
    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

MOV is used 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. A left shift by *n* can be used 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.
*   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 a MOVS PC, LR instruction can be used to return from some types of exception (see *Exceptions* on page A2-13).

4.1.30 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	
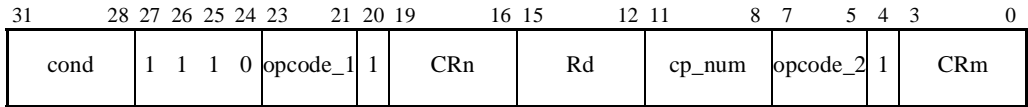
The MRC (Move to ARM Register from Coprocessor) instruction causes the coprocessor whose number is cp_num to transfer a value to an ARM register or to the condition flags.

If no coprocessors indicate that they 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 A3-5. 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 for the instruction.
<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

 ARM DDI 0100E

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>.

    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 trap.

4.1.31   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	
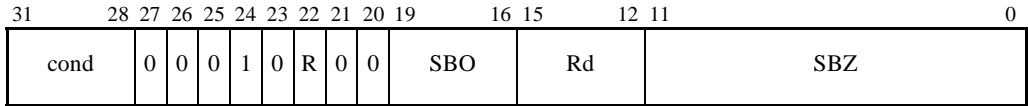
The MRS (Move PSR to General-purpose Register) instruction moves the value of the CPSR or the SPSR of the current mode into a general-purpose register. In the general-purpose register, the value can be examined or manipulated with normal data-processing instructions.

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 A3-5. If <cond> is omitted, the AL (always) condition is used.

<Rd>   Specifies the destination register of the instruction. 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

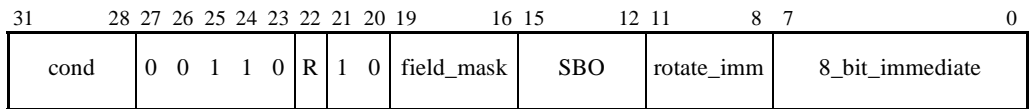The MRS instruction is commonly used for three purposes:

* As part of a read/modify/write sequence for updating a PSR. For more details, see *MSR* on page A4-62.

* 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. For an example of this usage, see *Interrupt prioritization* on page A9-15.

* In process swap code, the programmer's model state of the process being swapped out needs to be saved, including relevant PSR contents, and similar state of the process being swapped in needs to be restored. Again, this involves the use of MRS/store and load/MSR instruction sequences. For an example of this usage, see *Context switch* on page A9-16.
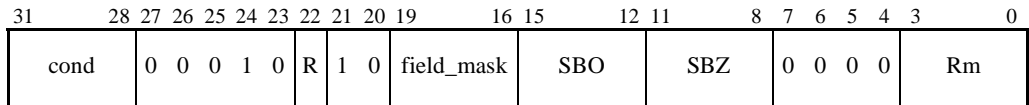
Notes

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.

        Copyright © 1996-2000 ARM Limited. All rights reserved.

4.1.32   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	

The MSR (Move to Status Register from ARM Register) instruction transfers the value of a general-purpose register or 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 A3-5. 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.

 ARM DDI 0100E

Architecture version

Version 3 and above

Exceptions

None

Operation

```
if ConditionPassed(cond) then
    if opcode[25] == 1
        operand = 8_bit_immediate Rotate_Right (rotate_imm * 2)
    else /* opcode[25] == 0 */
        operand = Rm
    if R == 0 then
        if field_mask[0] == 1 and InAPrivilegedMode() then
            CPSR[7:0] = operand[7:0]
        if field_mask[1] == 1 and InAPrivilegedMode() then
            CPSR[15:8] = operand[15:8]
        if field_mask[2] == 1 and InAPrivilegedMode() then
            CPSR[23:16] = operand[23:16]
        if field_mask[3] == 1 then
            CPSR[31:24] = operand[31:24]
    else /* R == 1 */
        if field_mask[0] == 1 and CurrentModeHasSPSR() then
            SPSR[7:0] = operand[7:0]
        if field_mask[1] == 1 and CurrentModeHasSPSR() then
            SPSR[15:8] = operand[15:8]
        if field_mask[2] == 1 and CurrentModeHasSPSR() then
            SPSR[23:16] = operand[23:16]
        if field_mask[3] == 1 and CurrentModeHasSPSR() then
            SPSR[31:24] = operand[31:24]
```

Usage

This instruction is used to update the value of the condition code flags, interrupt enables, or the processor mode.

The value of a PSR should normally be updated 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. For example, when the process swap code in *Context switch* on page A9-16 restores the new process's CPSR value, it writes to SPSR_fsxc (which is later copied to the CPSR by the LDM instruction which restarts the process). There are no bits allocated to the state or extension fields at present, so writing to SPSR_fc would work just as well in current architecture versions (5 and below). However, writing to SPSR_fsxc will continue to work correctly in future versions of the architecture that do have bits allocated in the state or extension fields, while writing to SPSR_fc will not.

———— **Note** ————

Due to deficiencies in the handling of the state and extension fields in versions 2.50 and below of the ARM Software Development Toolkit, only specify the flags and control fields when using these toolkit versions. In cases where the above guidelines mean that the state and extension fields should have been written for future compatibility, it is recommended that you keep a record of the need to change the set of fields specified when it becomes possible to do so.

The immediate form of this instruction can be used to set any of the fields of a PSR, but you must take care to adhere to 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[26:24] of the PSRs have no allocated function at present. For example, this instruction can be used 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 A4-62.

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   The MSR instruction must not be used to alter the T 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.

4.1.33   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	
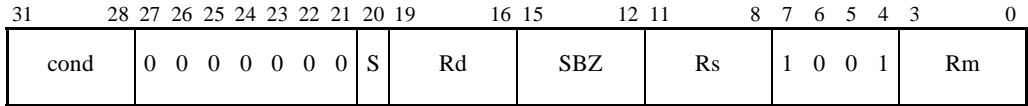
The MUL (Multiply) instruction is used to multiply signed or unsigned variables to produce a 32-bit result.
The condition code flags are optionally updated, based on the result.

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 A3-5. 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    /* See "C flag" note */
        V Flag = unaffected
```

Notes

Use of R15   Specifying R15 for register <Rd>, <Rm>, or <Rs> has UNPREDICTABLE results.

Operand restriction Specifying the same register for <Rd> and <Rm> 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.

4.1.34 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
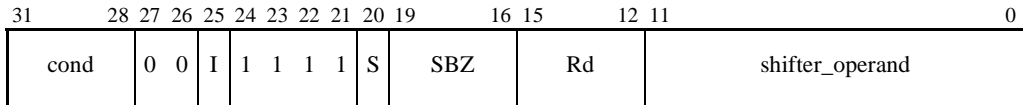
The MVN (Move Negative) instruction moves the logical one's complement of the value of
<shifter_operand> to the destination register <Rd>. The condition code flags are optionally updated,
based on the result.

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 A3-5. 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 of the instruction.

<shifter_operand>

            Specifies the operand for the operation. 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 A3-27 to determine which instruction it is.

Architecture version

All

Exceptions

None

                                 ARM DDI 0100E

Operation

```
if ConditionPassed(cond) then
    Rd = NOT shifter_operand
    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 = shifter_carry_out
        V Flag = unaffected
```
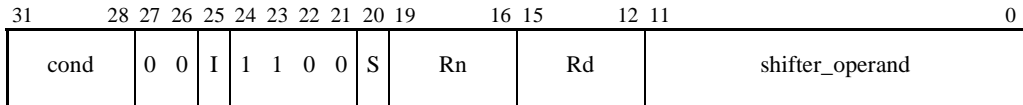
Usage

MVN is used to:

* write a negative value into a register
* form a bit mask
* take the one's complement of a value.

4.1.35   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	

The ORR (Logical OR) instruction performs a bitwise (inclusive) OR of the value of register <Rn> with the value of <shifter_operand>, and stores the result in the destination register <Rd>. The condition code flags are optionally updated, 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 A3-5. 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 of the instruction.

<Rn>   Specifies the register that contains the first operand for the operation.

<shifter_operand>

                Specifies the second operand for the operation. 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 A3-27 to determine which instruction it is.

Architecture version

All

Copyright © 1996-2000 ARM Limited. All rights reserved.

Exceptions

None

Operation

```
if ConditionPassed(cond) then
    Rd = Rn OR shifter_operand
    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 = shifter_carry_out
        V Flag = unaffected
```

Usage

ORR can be used to set selected bits in a register. For each bit, OR with 1 sets the bit, and OR with 0 leaves it unchanged.

4.1.36 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	
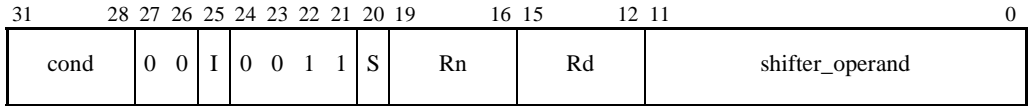
The RSB (Reverse Subtract) instruction subtracts the value of register <Rn> from the value of <shifter_operand>, and stores the result in the destination register <Rd>. The condition code flags are optionally updated, based on the result.

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 A3-5. 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 of the instruction.

<Rn>   Specifies the register that contains the second operand for the subtraction.

<shifter_operand>

                Specifies the first operand for the subtraction. 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 A3-27 to determine which instruction it is.

Architecture version

All

                   ARM DDI 0100E

Exceptions

None

Operation

```
if ConditionPassed(cond) then
    Rd = shifter_operand - Rn
    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 = 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
```

Constant multiplication (of `Rx`) by 2^n-1 (into `Rd`) can be performed 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.

4.1.37   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	
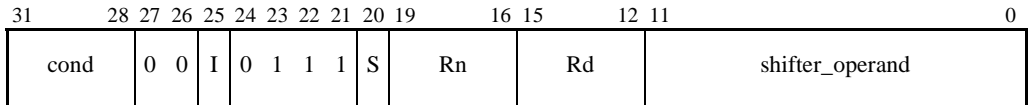
The `RSC` (Reverse Subtract with Carry) instruction subtracts the value of register `<Rn>` and the value of NOT(Carry flag) from the value of `<shifter_operand>`, and stores the result in the destination register `<Rd>`. The condition code flags are optionally updated, based on the result.

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 A3-5. 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 of the instruction.

<Rn>   Specifies the register that contains the second operand for the subtraction.

<shifter_operand>

   Specifies the first operand for the subtraction. 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 A3-27 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
        CPSR = SPSR
    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

To negate the 64-bit value in R0,R1, use the following sequence (R0 holds the least significant word) which stores the result in R2,R3:
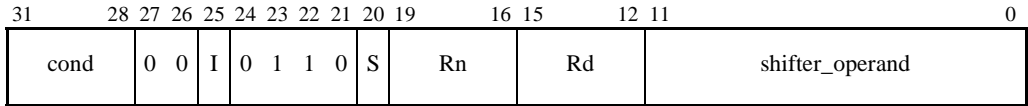
```
RSBS    R2,R0,#0
RSC     R3,R1,#0
```

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.

Copyright © 1996-2000 ARM Limited. All rights reserved.

4.1.38 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	

The SBC (Subtract with Carry) instruction is used to synthesize multi-word subtraction. SBC subtracts the value of <shifter_operand> and the value of NOT(Carry flag) from the value of register <Rn>, and stores the result in the destination register <Rd>. The condition code flags are optionally updated, 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 A3-5. 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 of the instruction.

<Rn>   Specifies the register that contains the first operand for the subtraction.

<shifter_operand>

   Specifies the second operand for the subtraction. 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 A3-27 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
        CPSR = SPSR
    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
```

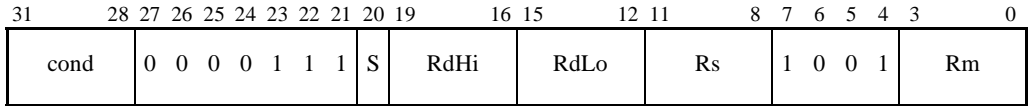## Notes

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 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.

4.1.39 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

The SMLAL (Signed Multiply Accumulate Long) instruction multiplies the signed value of register <Rm> with the signed 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

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 A3-5. 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

All M variants

Exceptions

None

         ARM DDI 0100E

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>, <RdLo>, and <Rm> must be three distinct registers, or the results are UNPREDICTABLE.
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 SMLALS instruction 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.

4.1.40   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
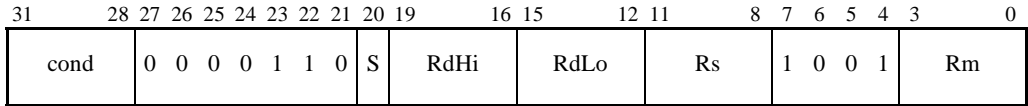
The SMULL (Signed Multiply Long) instruction multiplies the signed value of register <Rm> with the signed value of register <Rs> to produce a 64-bit result. The upper 32 bits of the result are stored in <RdHi>. The lower 32 bits are stored in <RdLo>. The condition code flags are optionally updated, based on the 64-bit result.

Syntax

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 A3-5. 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

All M variants

Exceptions

None

           ARM DDI 0100E

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 */
```
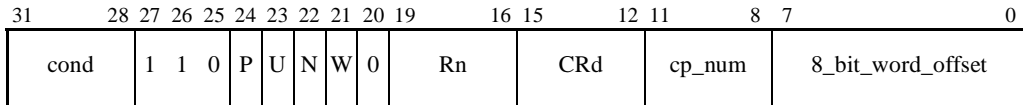
Usage

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>, <RdLo>, and <Rm> must be three distinct registers, or the results are
                        UNPREDICTABLE.

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 SMULLS instruction 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.

4.1.41   STC

cond	1 1 0	P	U	N	W	0	Rn	CRd	cp_num	8_bit_word_offset

31   28 27 26 25 24 23 22 21 20 19   16 15   12 11   8 7   0

The STC (Store Coprocessor) instruction stores data from the coprocessor whose name is cp_num to the sequence of consecutive memory addresses calculated by <addressing_mode>. 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 A3-5. 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 of the instruction.

<addressing_mode>

                Is described in *Addressing Mode 5 - Load and Store Coprocessor* on page A5-56. 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 writeback*).

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 A2-17.

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 B2 *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 trap.

4.1.42   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	
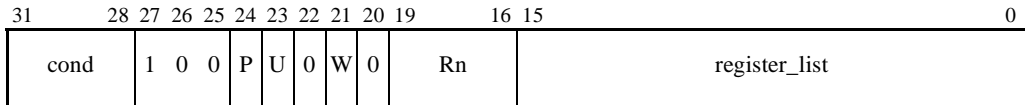
This form of the STM (Store Multiple) instruction stores a non-empty subset (or possibly all) of the general-purpose registers to sequential memory locations.

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 A3-5. If <cond> is omitted, the AL (always) condition is used.
<addressing_mode>	
	Is described in *Addressing Mode 4 - Load and Store Multiple* on page A5-48. 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 A5-48. 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 A2-7.

Architecture version

All

Exceptions

Data Abort

                       ARM DDI 0100E

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 writeback is specified:

- If <Rn> is the lowest-numbered register specified in <register_list>, the original value of <Rn> is stored.

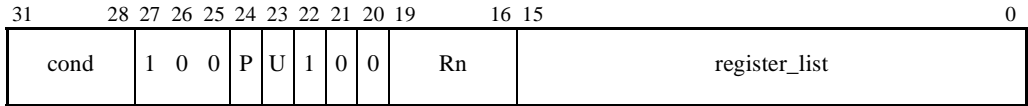- Otherwise, the stored value of <Rn> is UNPREDICTABLE.

Data abort   For details of the effects of the instruction if a data abort occurs, see *Effects of data-aborted instructions* on page A2-17.

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 B2 *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 A2-32 for details.

4.1.43   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	

This form of STM stores a subset (or possibly all) of the User mode 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 A3-5. If <cond> is omitted, the AL (always) condition is used.
<addressing_mode>	
	Is described in *Addressing Mode 4 - Load and Store Multiple* on page A5-48. 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 A2-7.
^	For an STM instruction, indicates that User mode registers are to be stored.

Architecture version

All

Exceptions

Data Abort

                   ARM DDI 0100E

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

STM is used to store the User mode registers when the processor is in a privileged mode (useful when performing process swaps, and in instruction emulators).

Notes

Banked registers   This instruction must not be followed by an instruction which accesses banked registers (a following NOP is a good way to ensure this).

Writeback   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.
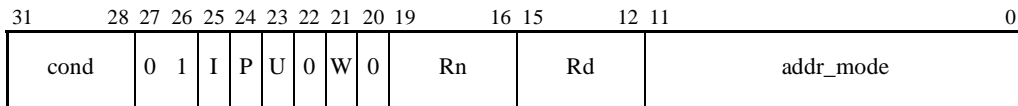
Data abort   For details of the effects of the instruction if a data abort occurs, see *Effects of data-aborted instructions* on page A2-17.

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 B2 *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 A2-32 for details.

4.1.44   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	

The STR (Store Register) instruction stores a word from register <Rd> to the memory address calculated by <addressing_mode>.

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 A3-5. 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 A2-7.

<addressing_mode>

Is described in *Addressing Mode 2 - Load and Store Word or Unsigned Byte* on page A5-18. 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 writeback*).

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 writeback, 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 A2-17.

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 B2 *The System Control Coprocessor*), and alignment checking is enabled, an address with bits[1:0] != 0b00 causes an alignment exception.

4.1.45   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	
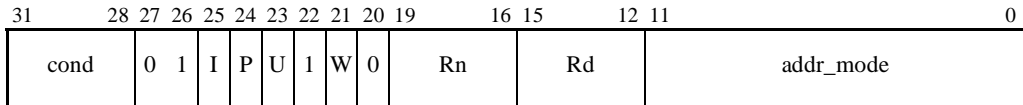
The STRB (Store Register Byte) instruction stores a byte from the least significant byte of register <Rd> to the memory address calculated by <addressing_mode>.

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 A3-5. 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 A5-18. 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 writeback*).

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 writeback, 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 A2-17.

4.1.46   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	
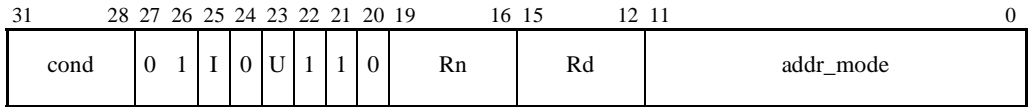
The STRBT (Store Register Byte with Translation) instruction stores a byte from the least significant byte of register <Rd> to the memory address calculated by <post_indexed_addressing_mode>. If the instruction is executed when the processor is in a privileged mode, the memory system is signaled to treat 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 A3-5. 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 A5-18. 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 writeback*).

Architecture version

All

Exceptions

Data Abort

Operation

```
if ConditionPassed(cond) then
    Memory[address,1] = Rd[7:0]
```

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.
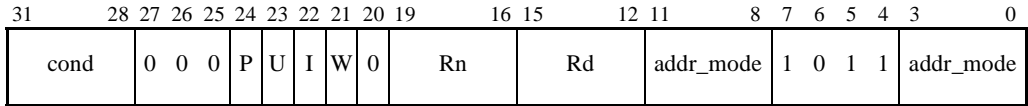
Notes

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 A2-17.

Copyright © 1996-2000 ARM Limited. All rights reserved.

4.1.47   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	

The STRH (Store Register Halfword) instruction stores a halfword from the least significant halfword of register <Rd> to the memory address calculated by <addressing_mode>. 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 A3-5. 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 A5-34. 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 writeback*).

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 writeback, 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 A2-17.

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 B2 *The System Control Coprocessor*), and alignment checking is enabled, an address with bit[0] != 0 causes an alignment exception.

4.1.48   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	
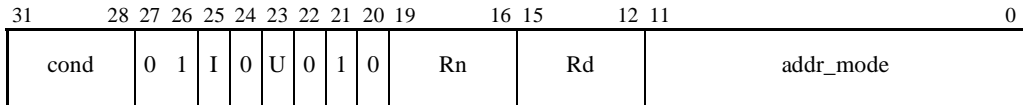
The STRT (Store Register with Translation) instruction stores a word from register <Rd> to the memory address calculated by <post_indexed_addressing_mode>. If the instruction is executed when the processor is in a privileged mode, the memory system is signaled to treat the access as if the processor 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 A3-5. 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 A2-7.

<post_indexed_addressing_mode>

Is described in *Addressing Mode 2 - Load and Store Word or Unsigned Byte* on page A5-18. 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 writeback*).

Architecture version

All

Exceptions

Data Abort

Operation

```
if ConditionPassed(cond) then
    Memory[address,4] = Rd
```

Copyright © 1996-2000 ARM Limited. All rights reserved.   ARM DDI 0100E

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.
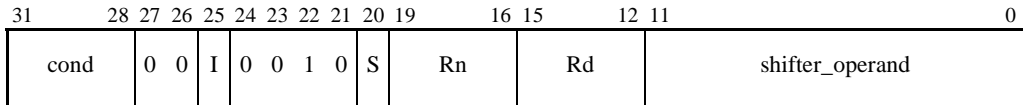
Notes

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 A2-17.

Alignment If an implementation includes a System Control coprocessor (see Chapter B2 *The System Control Coprocessor*), and alignment checking is enabled, an address with bits[1:0] != 0b00 causes an alignment exception.

4.1.49   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	

The SUB (Subtract) instruction subtracts the value of <shifter_operand> from the value of register
<Rn>, and stores the result in the destination register <Rd>. The condition code flags are optionally
updated, 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 A3-5. 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 of the instruction.

<Rn>   Specifies the register that contains the first operand for the subtraction.

<shifter_operand>

                Specifies the second operand for the subtraction. 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 A3-27 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
        CPSR = SPSR
    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

SUB is used to subtract one value from another to produce a third. To decrement a register value (in Rx) use:

```
    SUBS 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 compare instruction:

```
    CMP Rx, #0
```

This both decrements the loop counter in Ri and checks whether it has reached zero.

The form of this instruction with the PC as its destination register and the S bit set can be used to return from interrupts and various other types of exception. See *Exceptions* on page A2-13 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 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.

4.1.50 SWI

31	28 27 26 25 24	23 0
cond	1 1 1 1	immed_24
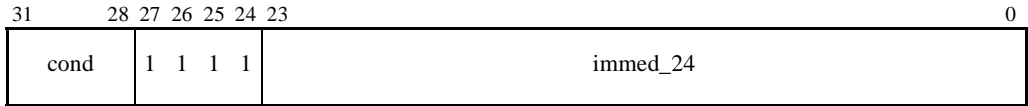
The SWI (Software Interrupt) instruction causes a SWI exception (see *Exceptions* on page A2-13).

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 A3-5. 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 A4-101 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
```

         ARM DDI 0100E

Usage

The SWI instruction 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.

4.1.51   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	

The SWP (Swap) instruction 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 A3-5. 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

The SWP instruction can be used to implement semaphores. For sample code, see *Semaphore instructions* on page A9-11.

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 B2 *The System Control Coprocessor*), and alignment checking is enabled, an address with bits[1:0] != 0b00 causes an alignment exception.

4.1.52 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	
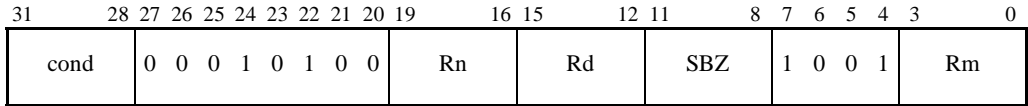
The SWPB (Swap Byte) instruction swaps a byte between registers and memory. SWPB loads a byte from the memory address given by the value of register <Rn>. The value of the least significant byte of register <Rm> is stored to the memory address given by <Rn>, the original loaded value is zero-extended 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 A3-5. 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
```
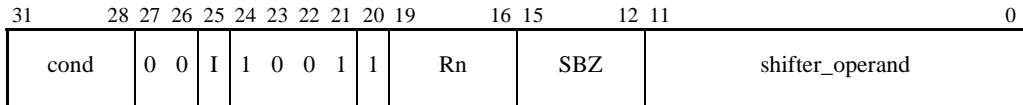
Usage

The SWPB instruction can be used to implement semaphores, in a similar manner to that shown for SWP instructions in *Semaphore instructions* on page A9-11.

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.

4.1.53 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	

The TEQ (Test Equivalence) instruction 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 A3-5. If <cond> is omitted, the AL (always) condition is used.

<Rn>   Specifies the register that contains the first operand for the comparison.

<shifter_operand>

Specifies the second operand for the comparison. 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 A3-27 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

TEQ is used 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.

4.1.54 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	
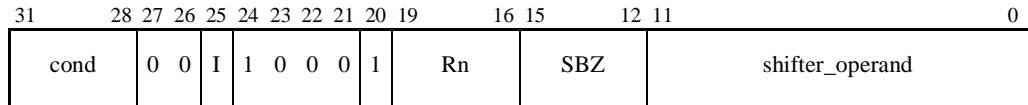
The TST (Test) instruction compares a register value with another arithmetic value. The condition flags are updated, based on the result of logically ANDing the two values, so that subsequent instructions can be conditionally executed.

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 A3-5. If <cond> is omitted, the AL (always) condition is used.

<Rn>   Specifies the register that contains the first operand for the comparison.

<shifter_operand>

Specifies the second operand for the comparison. 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 A3-27 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

TST is used 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.

 ARM DDI 0100E

4.1.55 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		

The `UMLAL` (Unsigned Multiply Accumulate Long) instruction 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 A3-5. 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

All M variants

Exceptions

None

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 */
```
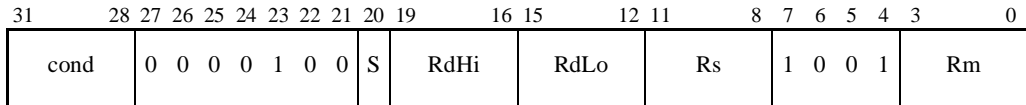
Usage

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>, <RdLo>, and <Rm> must be three distinct registers, or the results are UNPREDICTABLE.

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 UMLALS instruction 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.

4.1.56   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	

The UMULL (Unsigned Multiply Long) instruction 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 A3-5. 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

All M variants

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>, <RdLo>, and <Rm> must be three distinct registers, or the results are UNPREDICTABLE.

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 UMULLS instruction 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.

4.2 ARM instructions and architecture versions

Table 4-1 shows which ARM instructions are present in each current ARM architecture version.

Table 4-1 ARM instructions by architecture version

Instruction	v3, v3M	v4, v4xM	v4T, v4TxM	v5, v5xM, v5T, v5TxM	v5TE, v5TExP
ADC	Yes	Yes	Yes	Yes	Yes
ADD	Yes	Yes	Yes	Yes	Yes
AND	Yes	Yes	Yes	Yes	Yes
B	Yes	Yes	Yes	Yes	Yes
BIC	Yes	Yes	Yes	Yes	Yes
BKPT	No	No	No	Yes	Yes
BL	Yes	Yes	Yes	Yes	Yes
BLX (both forms)	No	No	No	Yes	Yes
BX	No	No	Yes	Yes	Yes
CDP	Yes	Yes	Yes	Yes	Yes
CDP2	No	No	No	Yes	Yes
CLZ	No	No	No	Yes	Yes
CMN	Yes	Yes	Yes	Yes	Yes
CMP	Yes	Yes	Yes	Yes	Yes
EOR	Yes	Yes	Yes	Yes	Yes
LDC	Yes	Yes	Yes	Yes	Yes
LDC2	No	No	No	Yes	Yes
LDM (all forms)	Yes	Yes	Yes	Yes	Yes
LDR	Yes	Yes	Yes	Yes	Yes
LDRB	Yes	Yes	Yes	Yes	Yes
LDRD	No	No	No	No	Only v5TE
LDRBT	Yes	Yes	Yes	Yes	Yes

Table 4-1 ARM instructions by architecture version *(Continued)*

Instruction	v3, v3M	v4, v4xM	v4T, v4TxM	v5, v5xM, v5T, v5TxM	v5TE, v5TExP
LDRH	No	Yes	Yes	Yes	Yes
LDRSB	No	Yes	Yes	Yes	Yes
LDRSH	No	Yes	Yes	Yes	Yes
LDRT	Yes	Yes	Yes	Yes	Yes
MCR	Yes	Yes	Yes	Yes	Yes
MCR2	No	No	No	Yes	Yes
MCRR	No	No	No	No	Only v5TE
MLA	Yes	Yes	Yes	Yes	Yes
MOV	Yes	Yes	Yes	Yes	Yes
MRC	Yes	Yes	Yes	Yes	Yes
MRC2	No	No	No	Yes	Yes
MRRC	No	No	No	No	Only v5TE
MRS	Yes	Yes	Yes	Yes	Yes
MSR	Yes	Yes	Yes	Yes	Yes
MUL	Yes	Yes	Yes	Yes	Yes
MVN	Yes	Yes	Yes	Yes	Yes
ORR	Yes	Yes	Yes	Yes	Yes
PLD	No	No	No	No	Only v5TE
QADD	No	No	No	No	Yes
QDADD	No	No	No	No	Yes
QDSUB	No	No	No	No	Yes
QSUB	No	No	No	No	Yes
RSB	Yes	Yes	Yes	Yes	Yes
RSC	Yes	Yes	Yes	Yes	Yes

Table 4-1 ARM instructions by architecture version *(Continued)*

Instruction	v3, v3M	v4, v4xM	v4T, v4TxM	v5, v5xM, v5T, v5TxM	v5TE, v5TExP
SBC	Yes	Yes	Yes	Yes	Yes
SMLAL	Only v3M	Only v4	Only v4T	Only v5/v5T	Yes
SMLA<x><y>	No	No	No	No	Yes
SMLAL<x><y>	No	No	No	No	Yes
SMLAW<y>	No	No	No	No	Yes
SMULL	Only v3M	Only v4	Only v4T	Only v5/v5T	Yes
SMUL<x><y>	No	No	No	No	Yes
SMULW<y>	No	No	No	No	Yes
STC	Yes	Yes	Yes	Yes	Yes
STC2	No	No	No	Yes	Yes
STM (both forms)	Yes	Yes	Yes	Yes	Yes
STR	Yes	Yes	Yes	Yes	Yes
STRB	Yes	Yes	Yes	Yes	Yes
STRBT	Yes	Yes	Yes	Yes	Yes
STRD	No	No	No	No	Only v5TE
STRH	No	Yes	Yes	Yes	Yes
STRT	Yes	Yes	Yes	Yes	Yes
SUB	Yes	Yes	Yes	Yes	Yes
SWI	Yes	Yes	Yes	Yes	Yes
SWP	Yes	Yes	Yes	Yes	Yes
SWPB	Yes	Yes	Yes	Yes	Yes
TEQ	Yes	Yes	Yes	Yes	Yes

Table 4-1 ARM instructions by architecture version *(Continued)*

Instruction	v3, v3M	v4, v4xM	v4T, v4TxM	v5, v5xM, v5T, v5TxM	v5TE, v5TExP
TST	Yes	Yes	Yes	Yes	Yes
UMLAL	Only v3M	Only v4	Only v4T	Only v5/v5T	Yes
UMULL	Only v3M	Only v4	Only v4T	Only v5/v5T	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:

5.1   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 A5-15.

10.   `<Rm>, ROR <Rs>`

    See *Data-processing operands - Rotate right by register* on page A5-16.

11.   `<Rm>, RRX`

    See *Data-processing operands - Rotate right with extend* on page A5-17.

5.1.1 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
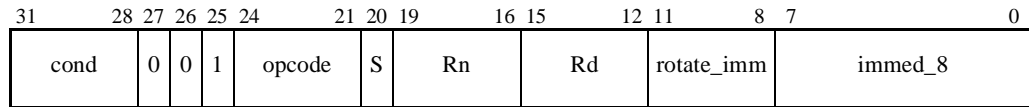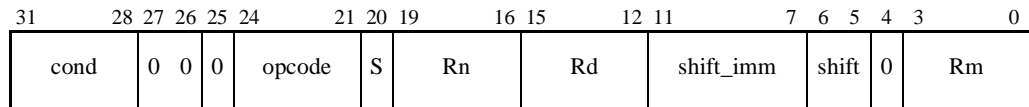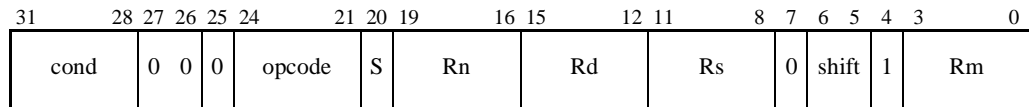
opcode   Specifies the operation of the instruction.

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 A3-27 for more information.

5.1.2 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.

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
```

5.1.3   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	
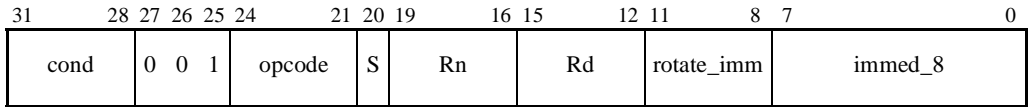
This data-processing operand provides a constant (defined in the instruction) operand to a data-processing instruction.

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

When more than one encoding is available, an assembler needs to 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.

5.1.4   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	
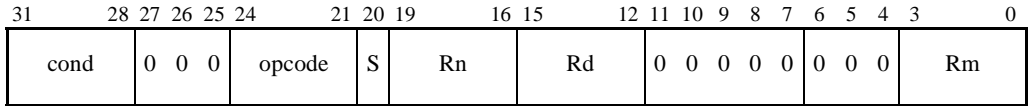
This data-processing operand provides the value of a register directly. The carry-out from the shifter is the C flag.

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.

5.1.5   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.

5.1.6 Data-processing operands - Logical shift left by register
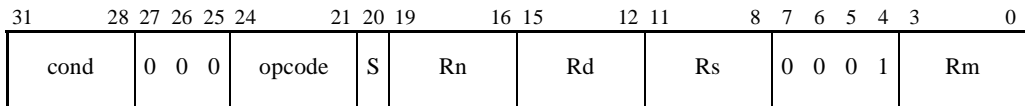
31	28	27 26 25	24	21	20	19	16	15	12	11	8	7 6 5 4	3	0
cond		0 0 0	opcode		S	Rn		Rd		Rs		0 0 0 1	Rm	

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.

5.1.7   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.

5.1.8 Data-processing operands - Logical shift right by register
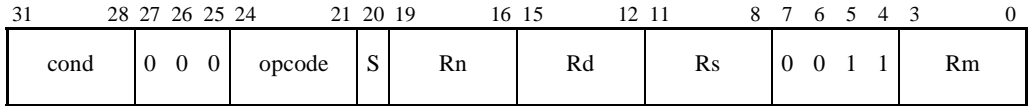
31	28 27 26 25	24 21	20	19 16	15 12	11 8	7 6 5 4	3 0
cond	0 0 0	opcode	S	Rn	Rd	Rs	0 0 1 1	Rm

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.

Copyright © 1996-2000 ARM Limited. All rights reserved.   ARM DDI 0100E

5.1.9   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.

5.1.10   Data-processing operands - Arithmetic shift right by register
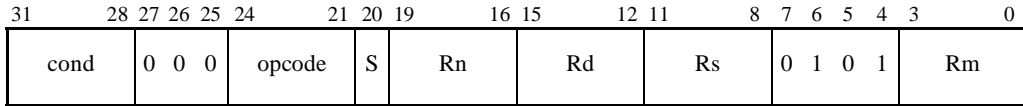
31	28 27 26 25	24	21 20	19	16 15	12 11	8	7 6 5 4	3	0
cond	0 0 0	opcode	S	Rn	Rd	Rs		0 1 0 1	Rm	

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.

           ARM DDI 0100E

5.1.11 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 A5-17.

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.

5.1.12 Data-processing operands - Rotate right by register
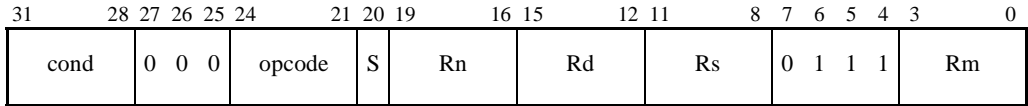
31	28	27 26 25	24	21	20	19	16	15	12	11	8	7 6 5 4	3	0
cond		0 0 0	opcode		S	Rn		Rd		Rs		0 1 1 1	Rm	

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.

5.1.13   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.

5.2   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 A10-14, 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 A5-20.

2. `[<Rn>, +/-<Rm>]`

    See *Load and Store Word or Unsigned Byte - Register offset* on page A5-21.

3. `[<Rn>, +/-<Rm>, <shift> #<shift_imm>]`

    See *Load and Store Word or Unsigned Byte - Scaled register offset* on page A5-22.

4. `[<Rn>, #+/-<offset_12>]!`

    See *Load and Store Word or Unsigned Byte - Immediate pre-indexed* on page A5-24.

5. `[<Rn>, +/-<Rm>]!`

    See *Load and Store Word or Unsigned Byte - Register pre-indexed* on page A5-25.

6. `[<Rn>, +/-<Rm>, <shift> #<shift_imm>]!`

    See *Load and Store Word or Unsigned Byte - Scaled register pre-indexed* on page A5-26.

7. `[<Rn>], #+/-<offset_12>`

    See *Load and Store Word or Unsigned Byte - Immediate post-indexed* on page A5-28.

8. `[<Rn>], +/-<Rm>`

    See *Load and Store Word or Unsigned Byte - Register post-indexed* on page A5-30.

9. `[<Rn>], +/-<Rm>, <shift> #<shift_imm>`

    See *Load and Store Word or Unsigned Byte - Scaled register post-indexed* on page A5-32.

                                       ARM DDI 0100E

5.2.1 Encoding

The following three diagrams show the encodings for this addressing mode:
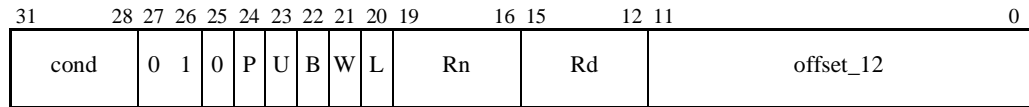
Immediate offset/index

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).

5.2.2 Load and Store Word or Unsigned Byte - Immediate offset
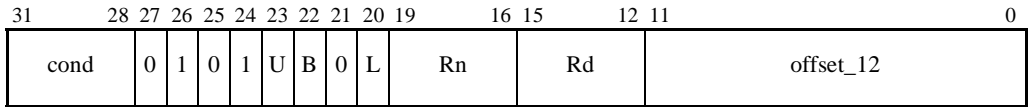
31 28	27	26	25	24	23	22	21	20	19 16	15 12	11 0
cond	0	1	0	1	U	B	0	L	Rn	Rd	offset_12

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.

5.2.3 Load and Store Word or Unsigned Byte - Register offset
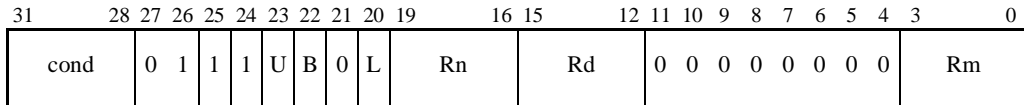
31	28	27	26	25	24	23	22	21	20	19	16	15	12	11	10	9	8	7	6	5	4	3	0
cond		0	1	1	1	U	B	0	L	Rn		Rd		0	0	0	0	0	0	0	0	Rm	

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.

5.2.4   Load and Store Word or Unsigned Byte - Scaled register offset
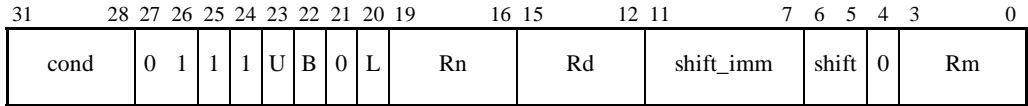
31 28	27 26 25 24	23	22	21	20	19 16	15 12	11 7	6 5	4	3 0
cond	0 1 1 1	U	B	0	L	Rn	Rd	shift_imm	shift	0	Rm

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
```
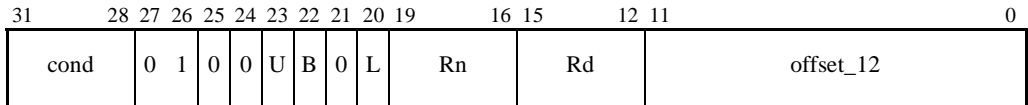
Usage

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.

5.2.5 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 A3-5.

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.

 ARM DDI 0100E

5.2.6   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 A3-5.

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	If the same register is specified for Rn and Rm, the result is UNPREDICTABLE.

5.2.7   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 A3-5.

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.

                    ARM DDI 0100E

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   If the same register is specified for Rn and Rm, the result is UNPREDICTABLE.

5.2.8   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 A3-5.

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.

5.2.9 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 A3-5.

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   If the same register is specified for Rn and Rm, the result is UNPREDICTABLE.

5.2.10   Load and Store Word or Unsigned Byte - Scaled register post-indexed
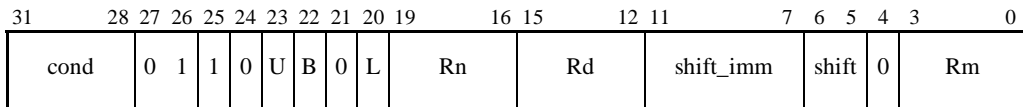
31 28	27 26 25 24	23	22	21	20	19 16	15 12	11 7	6 5	4	3 0
cond	0 1 1 0	U	B	0	L	Rn	Rd	shift_imm	shift	0	Rm

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 A3-5.

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.)

 ARM DDI 0100E

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

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 Rm or Rn has UNPREDICTABLE results.

Operand restriction If the same register is specified for Rn and Rm, the result is UNPREDICTABLE.

5.3 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 A5-36.

2. `[<Rn>, +/-<Rm>]`

    See *Miscellaneous Loads and Stores - Register offset* on page A5-38.

3. `[<Rn>, #+/-<offset_8>]!`

    See *Miscellaneous Loads and Stores - Immediate pre-indexed* on page A5-40.

4. `[<Rn>, +/-<Rm>]!`

    See *Miscellaneous Loads and Stores - Register pre-indexed* on page A5-42.

5. `[<Rn>], #+/-<offset_8>`

    See *Miscellaneous Loads and Stores - Immediate post-indexed* on page A5-44.

6. `[<Rn>], +/-<Rm>`

    See *Miscellaneous Loads and Stores - Register post-indexed* on page A5-46.

5.3.1 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

 ARM DDI 0100E

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 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 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 A3-27).

    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 A3-27).

    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.

     Copyright © 1996-2000 ARM Limited. All rights reserved.

5.3.2   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.

    ARM DDI 0100E

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 A3-27).

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 A3-27).

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.

5.3.3 Miscellaneous Loads and Stores - Register offset
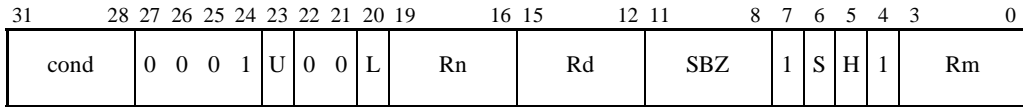
31	28	27	26	25	24	23	22	21	20	19	16	15	12	11	8	7	6	5	4	3	0
cond		0	0	0	1	U	0	0	L	Rn		Rd		SBZ		1	S	H	1	Rm	

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 A3-27).

Unsigned bytes are accessed by the LDRB, LDRBT, STRB and STRBT instructions, which use addressing mode 2 rather than addressing mode 3.

               ARM DDI 0100E

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 A3-27).

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.

5.3.4 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 A3-5.

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 A3-27).
	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 A3-27).
	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.

5.3.5 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 A3-5.

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 A3-27).

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 A3-27).

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   If the same register is specified for Rn and Rm, the result is UNPREDICTABLE.

5.3.6   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 A3-5.

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.

         ARM DDI 0100E

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 A3-27).
	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 A3-27).
	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.

5.3.7 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 A3-5.

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 A3-27).

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 A3-27).

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   If the same register is specified for Rn and Rm, the result is UNPREDICTABLE.

5.4 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 A5-50.

2. `IB` (Increment Before)

    See *Load and Store Multiple - Increment before* on page A5-51.

3. `DA` (Decrement After)

    See *Load and Store Multiple - Decrement after* on page A5-52.

4. `DB` (Decrement Before)

    See *Load and Store Multiple - Decrement before* on page A5-53.

5.4.1 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
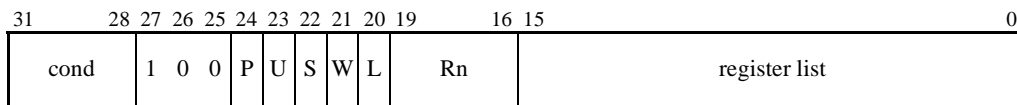
The P bit   Has two meanings:

  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 `LDM`s that load the PC, the S bit indicates that the CPSR is loaded from the SPSR. For `LDM`s that do not load the PC and all `STM`s, 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.

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 }.

5.4.2   Load and Store Multiple - Increment after
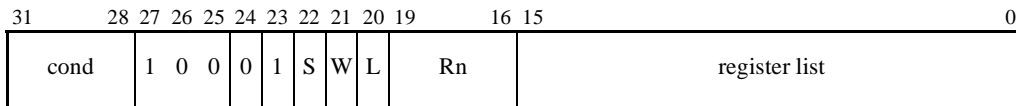
31	28	27	26	25	24	23	22	21	20	19	16	15	0
cond		1	0	0	0	1	S	W	L	Rn		register list	

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 A3-5.

Syntax

```
IA
```

See also the alternative syntax described in *Load and Store Multiple addressing modes (alternative names)* on page A5-54.

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.

5.4.3 Load and Store Multiple - Increment before
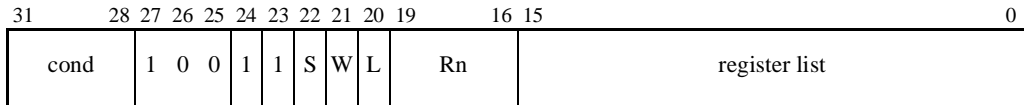
31	28	27 26 25	24	23	22	21	20	19 16	15 0
cond		1 0 0	1	1	S	W	L	Rn	register list

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 A3-5.

Syntax

```
IB
```

See also the alternative syntax described in *Load and Store Multiple addressing modes (alternative names)* on page A5-54.

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.

5.4.4   Load and Store Multiple - Decrement after
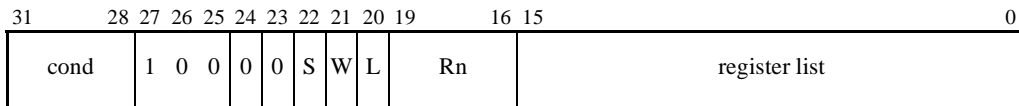
31	28	27	26	25	24	23	22	21	20	19	16	15	0
cond		1	0	0	0	0	S	W	L	Rn		register list	

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 A3-5.

Syntax

`DA`

See also the alternative syntax described in *Load and Store Multiple addressing modes (alternative names)* on page A5-54.

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 `LDM`s that load the PC, the S bit indicates that the CPSR is loaded from the SPSR. For `LDM`s that do not load the PC and all `STM`s, 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.

5.4.5   Load and Store Multiple - Decrement before
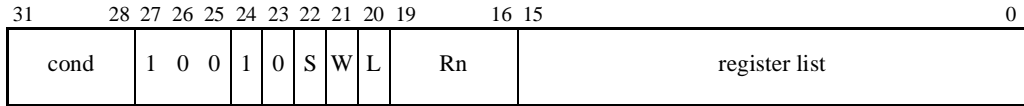
31	28	27 26 25	24	23	22	21	20	19	16	15	0
cond		1 0 0	1	0	S	W	L	Rn		register list	

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 A3-5.

Syntax

DB

See also the alternative syntax described in *Load and Store Multiple addressing modes (alternative names)* on page A5-54.

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.

5.4.6 Load and Store Multiple addressing modes (alternative names)

The four addressing mode names given in *Addressing Mode 4 - Load and Store Multiple* on page A5-48 (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 5-1 and Table 5-2 on page A5-55 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 5-1 shows the relationship for `LDM` instructions.

Table 5-1 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 5-2 shows the relationship for `STM` instructions.

Table 5-2 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

5.5 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 A5-58.

2.   `[<Rn>,#+/-<offset_8>*4]!`

   See *Load and Store Coprocessor - Immediate pre-indexed* on page A5-60.

3.   `[<Rn>],#+/-<offset_8>*4`

   See *Load and Store Coprocessor - Immediate post-indexed* on page A5-62.

4.   `[<Rn>],<option>`

   See *Load and Store Coprocessor - Unindexed* on page A5-64.

5.5.1 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
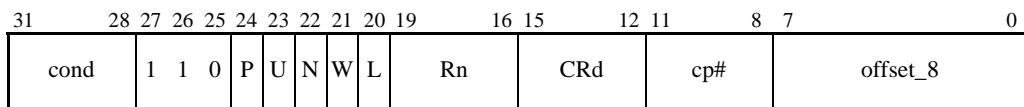
The P bit   Has two meanings:

   P == 1   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 == 0   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 he 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 A3-33).

- 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.

5.5.2   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 0100E

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.

5.5.3 Load and Store Coprocessor - Immediate pre-indexed
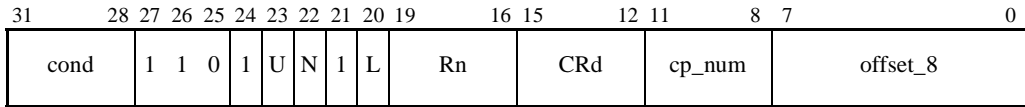
31	28	27	26	25	24	23	22	21	20	19	16	15	12	11	8	7	0
cond		1	1	0	1	U	N	1	L		Rn		CRd		cp_num		offset_8

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.

5.5.4   Load and Store Coprocessor - Immediate post-indexed
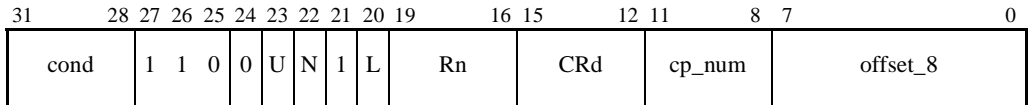
31	28	27 26 25 24	23	22	21	20	19	16	15	12	11	8	7	0
cond		1 1 0 0	U	N	1	L		Rn		CRd	cp_num		offset_8	

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
```

                   ARM DDI 0100E

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.

5.5.5　Load and Store Coprocessor - Unindexed
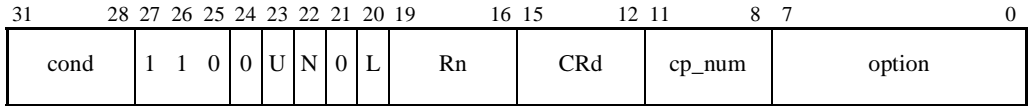
31	28	27	26	25	24	23	22	21	20	19	16	15	12	11	8	7	0
cond		1	1	0	0	U	N	0	L	Rn		CRd		cp_num		option	

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 A3-33).
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:

6.1 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 ARM. 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 know 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 does not provide direct access to the CPSR or any SPSR (as the MSR and MRS instructions do in the ARM instruction set)). 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.

──── **Note** ────

The Thumb instruction set is only compatible with the 32-bit ARM architectures. Thumb is not recommended for use with 26-bit architectures or with 26-bit compatibility options on 32-bit architectures.

6.1.1 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.

6.1.2   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 6-1 lists the values of the exception mode LR for exceptions generated during Thumb execution.

Table 6-1 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 A2-13. 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 A2-17, 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 A2-15, 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 in order to determine which of the two return instructions to use.

6.2 Instruction set encoding

Figure 6-1 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 6-1 Thumb instruction set overview

 ARM DDI 0100E

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.

6.2.1 Miscellaneous instructions

Figure 6-2 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				
Push/pop register list	1	0	1	1	L	1	0	R				register list				
Software breakpoint [1]	1	0	1	1	1	1	1	0				immediate				

Figure 6-2 Miscellaneous Thumb instructions

1. This is an undefined instruction prior to ARM architecture version 5.

——— **Note** ———

Any instruction with bits[15:12] = 1011, and which is not shown in Figure 6-2, is an undefined instruction.

6.3 Branch instructions

Thumb supports six types of branch instruction:

- an unconditional branch that allows a forward or backward branch of up to 2KB

- a conditional branch to allow forward and backward branches of up to 256 bytes

- 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 and Exchange instruction branches to an address in a register and optionally switches to ARM code execution

- a Branch with Link and Exchange instruction performs a subroutine call to an address in a register and optionally switches to ARM code execution

- a second form of Branch with Link and Exchange uses a pair of instructions, similar to Branch with Link, but additionally switches to ARM code execution.

The encoding for these instructions is given below.

6.3.1 Conditional branch

```
B<cond>   <target_address>
```

15	14	13	12	11 8	7 0
1	1	0	1	cond	8_bit_signed_offset
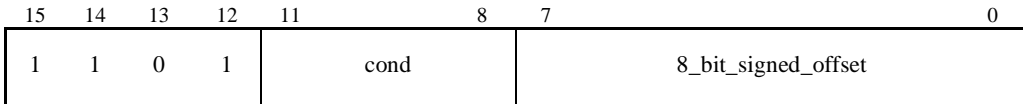
6.3.2 Unconditional branch

```
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
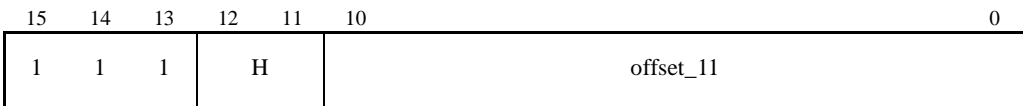
6.3.3 Branch with exchange

```
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
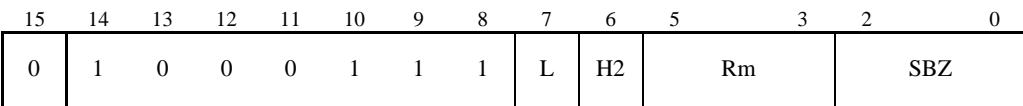
6.3.4   Examples

```
    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
```

6.3.5   List of branch instructions

The following instructions follow the formats shown above.

B   Conditional Branch. See *B (1)* on page A7-18.

B   Unconditional Branch. See *B (2)* on page A7-20.

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.

6.4   Data-processing instructions

Thumb data-processing instructions are a subset of the ARM data-processing instructions, as shown in Table 6-2. All Thumb data-processing instructions in this table set the condition codes.

Table 6-2 Thumb 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
`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
`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

Table 6-2 Thumb data-processing instructions (Continued)

Mnemonic	Operation	Action
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
```

6.4.1   High registers

There are seven types of data-processing instruction which operate on ARM registers 8 to 14 and the PC as shown in Table 6-3. Apart from CMP, instructions in this table do not change the condition code flags.

Table 6-3 High register data-processing instructions

Mnemonic	Operation	Action
MOV Rd, Rn	Move	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, R2          ; R6 = R1 - R2

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
```

6.4.2   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		
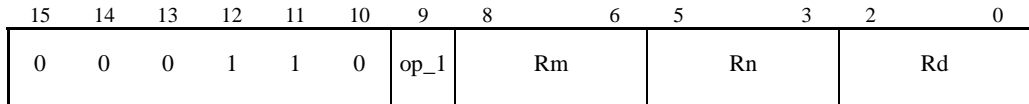
Format 2

```
<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		
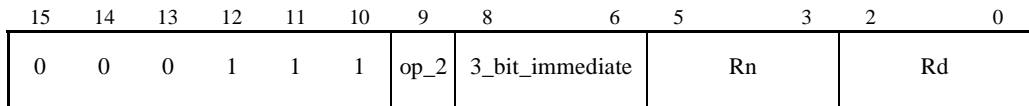
Format 3

```
<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		
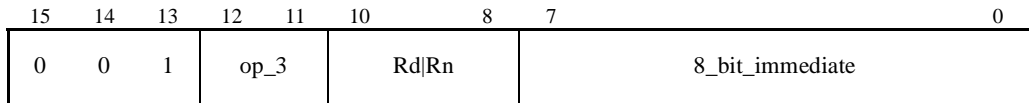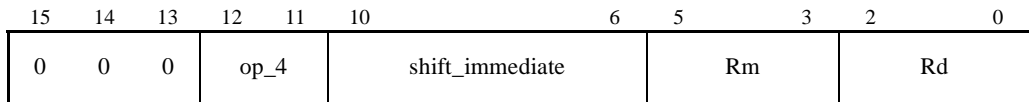
Format 5

```
<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	
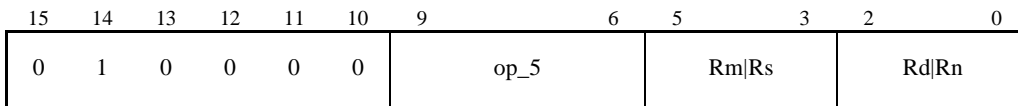
Format 6

```
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				
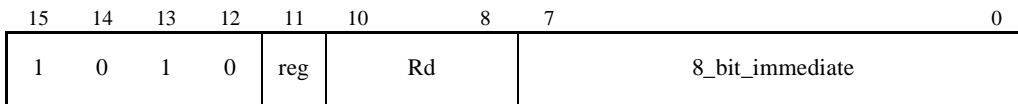
Format 7

```
<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
```

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		
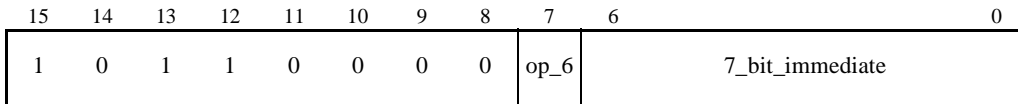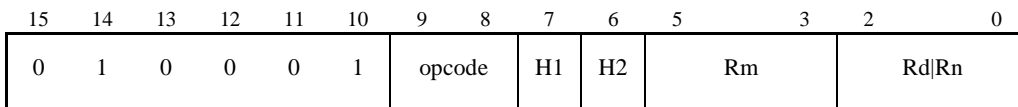
6.4.3 List of data-processing instructions

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 A7-13.
ASR	Arithmetic Shift Right (immediate). See *ASR (1)* on page A7-14.
ASR	Arithmetic Shift Right (register). See *ASR (2)* on page A7-16.
BIC	Bit Clear. See *BIC* on page A7-22.
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.
EOR	Exclusive OR. See *EOR* on page A7-39.
LSL	Logical Shift Left (immediate). See *LSL (1)* on page A7-59.
LSL	Logical Shift Left (register). See *LSL (2)* on page A7-60.
LSR	Logical Shift Right (immediate). See *LSR (1)* on page A7-62.
LSR	Logical Shift Right (register). See *LSR (2)* on page A7-64.
MOV	Move (immediate). See *MOV (1)* on page A7-66.
MOV	Move a low register to another low register. See *MOV (2)* on page A7-67.
MOV	Move high registers. See *MOV (3)* on page A7-68.
MUL	Multiply. See *MUL* on page A7-70.
MVN	Move NOT (register). See *MVN* on page A7-72.
NEG	Negate (register). See *NEG* on page A7-73.

ORR   Logical OR. See *ORR* on page A7-74.

ROR   Rotate Right (register). See *ROR* on page A7-80.

SBC   Subtract with Carry (register). See *SBC* on page A7-82.

SUB   Subtract (immediate). See *SUB (1)* on page A7-98.

SUB   Subtract (large immediate). See *SUB (2)* on page A7-99.

SUB   Subtract (register). See *SUB (3)* on page A7-100.

SUB   Decrement stack pointer. See *SUB (4)* on page A7-101.

TST   Test (register). See *TST* on page A7-103.

                   ARM DDI 0100E

6.5 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.

6.5.1 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
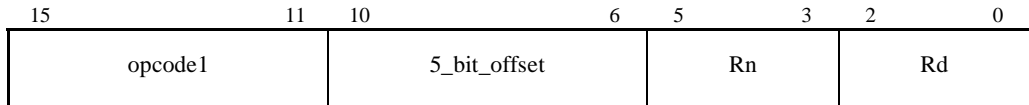
Format 2

```
<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
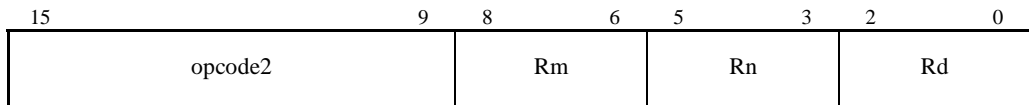
Format 3

```
LDR   <Rd>, [PC, #<8_bit_offset>]
```

15	14	13	12	11	10 8	7 0
0	1	0	0	1	Rd	8_bit_immediate

Copyright © 1996-2000 ARM Limited. All rights reserved.
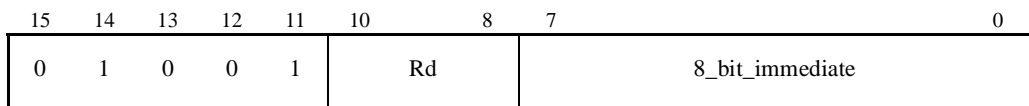
Format 4

```
<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	
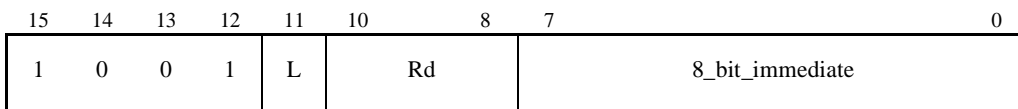
For example:

```
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
```

6.5.2 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 A7-42.

LDR   Load Word (register offset). See *LDR (2)* on page A7-44.

LDR   Load Word (PC-relative). See *LDR (3)* on page A7-46.

LDR   Load Word (SP-relative). See *LDR (4)* on page A7-48.

LDRB   Load Unsigned Byte (immediate offset). See *LDRB (1)* on page A7-50.

LDRB   Load Unsigned Byte (register offset). See *LDRB (2)* on page A7-51.

LDRH   Load Unsigned Halfword (immediate offset). See *LDRH (1)* on page A7-52.

LDRH   Load Unsigned Halfword (register offset). See *LDRH (2)* on page A7-54.

LDRSB   Load Signed Byte (register offset). See *LDRSB* on page A7-56.

LDRSH   Load Signed Halfword (register offset). See *LDRSH* on page A7-57.

STR   Store Word (immediate offset). See *STR (1)* on page A7-86.

STR   Store Word (register offset). See *STR (2)* on page A7-88.

STR   Store Word (SP-relative). See *STR (3)* on page A7-90.

STRB   Store Byte (immediate offset). See *STRB (1)* on page A7-92.

STRB   Store Byte (register offset). See *STRB (2)* on page A7-93.

STRH   Store Halfword (immediate offset). See *STRH (1)* on page A7-94.

STRH   Store Halfword (register offset). See *STRH (2)* on page A7-96.

6.6   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.

6.6.1   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	
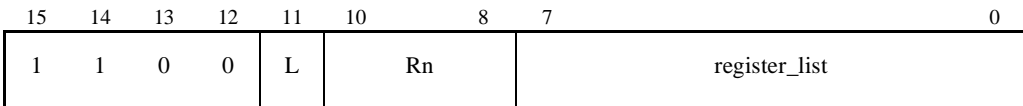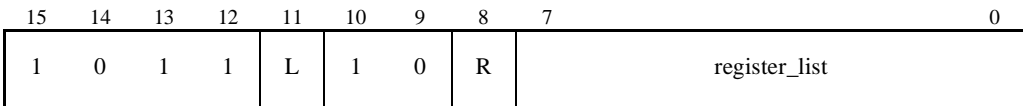
6.6.2   Examples

```
    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
```

     ARM DDI 0100E

6.6.3 List of Load and Store Multiple instructions

The following instructions follow the formats shown above.

LDMIA   Load Multiple. See *LDMIA* on page A7-40.

POP   Pop Multiple. See *POP* on page A7-75.

PUSH   Push Multiple. See *PUSH* on page A7-78.

STMIA   Store Multiple. See *STMIA* on page A7-84.

6.7 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 A2-16). 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 A2-16). 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.

6.7.1 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.

6.7.2 List of exception-generating instructions

BKPT   Breakpoint. See *BKPT* on page A7-24.

SWI   Software Interrupt. See *SWI* on page A7-102.

                           ARM DDI 0100E

6.8 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	x	x	x	x	x	x	x	x	x
1	0	1	1	1	0	x	x	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 A7-104.

7.1 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.

7.1.1 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 instruction set that the instruction is associated with, not the version of the Thumb instruction set. There have been two 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 and above of the ARM instruction set architecture.

Instructions which are described as being in all T variants are therefore present in both THUMBv1 and THUMBv2, while those that are described as being in T variants of version 5 and above are in THUMBv2 only.

        ARM DDI 0100E

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.

7.1.2   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	

The ADC (Add with Carry) instruction can be used to synthesize multi-word addition. The condition code flags are updated, 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		

7.1.3 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		

This form of ADD adds a small constant value to the value of a register and stores the result in a second register. The condition code flags are updated, 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		

7.1.4 ADD (2)

15	14	13	12	11	10		8	7		0
0	0	1	1	0		Rd			immed_8	
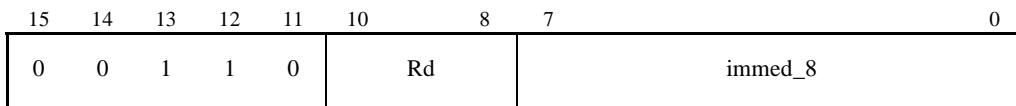
This form of ADD adds a large immediate value to the value of a register and stores the result back in the same register. The condition code flags are updated, based on the result.

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	
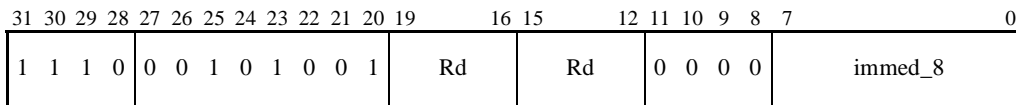
7.1.5   ADD (3)

15	14	13	12	11	10	9	8	6	5	3	2	0
0	0	0	1	1	0	0		Rm		Rn		Rd

This form of ADD adds the value of one register to the value of a second register, and stores the result in a third register. The condition code flags are updated, 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

7.1.6 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		

This form of ADD adds the values of two registers, one or both of which are high registers. Unlike the low-register 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.

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.

7.1.7   ADD (5)

15	14	13	12	11	10		8	7				0
1	0	1	0	0		Rd				immed_8		
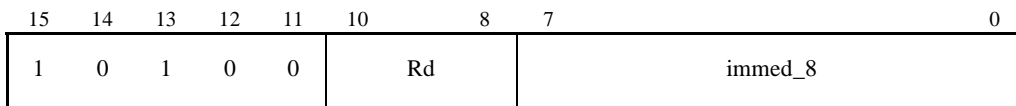
This form of ADD adds an immediate value to the PC and writes the resulting PC-relative address to a destination register. The immediate can be any multiple of 4 in the range 0 to 1020. The condition codes are not affected.

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		
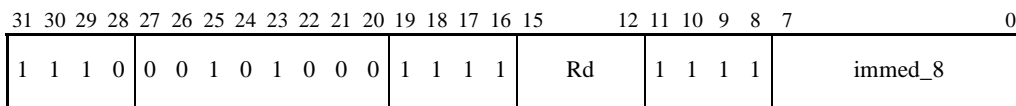
The definitions of the PC differ between ARM and Thumb code. This makes a difference between the precise results of the instructions.

7.1.8   ADD (6)

15	14	13	12	11	10 8	7 0
1	0	1	0	1	Rd	immed_8

This form of ADD 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

7.1.9   ADD (7)

15	14	13	12	11	10	9	8	7	6	0
1	0	1	1	0	0	0	0	0	immed_7	
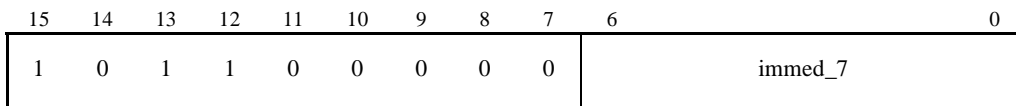
This form of ADD increments the SP by four times a 7-bit immediate (that is, by a multiple of 4 in the range 0 to 508). The condition codes are not affected.

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
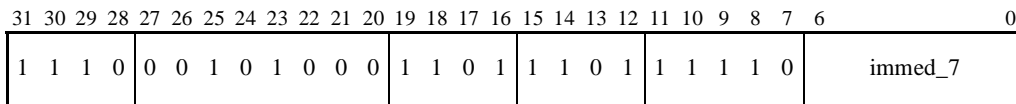
7.1.10 AND

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	

The AND (Logical AND) instruction performs a bitwise AND of the values in two registers. The condition code flags are updated, 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		

Copyright © 1996-2000 ARM Limited. All rights reserved.

7.1.11   ASR (1)

15	14	13	12	11	10	6	5	3	2	0
0	0	0	1	0	immed_5		Rm		Rd	
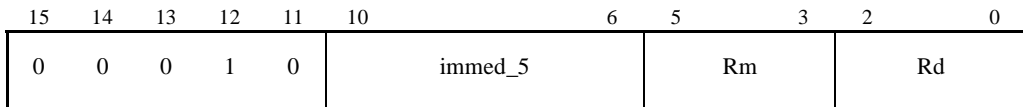
This form of the ASR (Arithmetic Shift Right) instruction is used to provide the signed value of a register divided by a constant power of 2. The condition code flags are updated, based on the result.

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

7.1.12   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	

This form of ASR is used to provide the signed value of a register divided by a variable power of 2. The condition code flags are updated, 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

7.1.13   B (1)

15	14	13	12	11				8	7							0
1	1	0	1			cond						signed_immed_8				
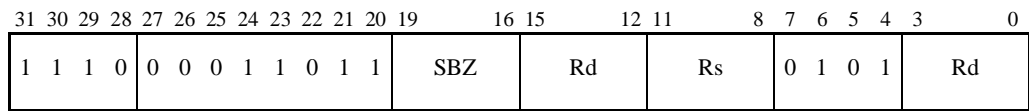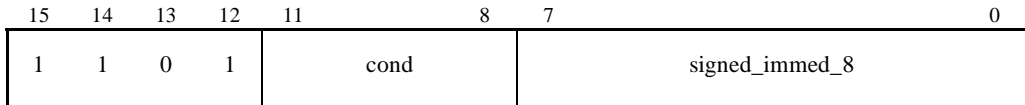
This form of the B (Branch) instruction provides a conditional branch to a target address.

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 A3-5.

<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 approximately ±256 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) needs to:

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 A7-20.

NV condition   If the condition field indicates NV (0b1111), the instruction is a SWI instead (see *SWI* on page A7-102).

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.

7.1.14   B (2)

15	14	13	12	11	10	0
1	1	1	0	0	signed_immed_11	

This form of B 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 approximately ±2048 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) needs to:

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.

7.1.15   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	

The BIC (Bit Clear) instruction performs a bitwise AND of the value of one register and the bitwise inverse of the value of another register. The condition code flags are updated, 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

7.1.16   BKPT
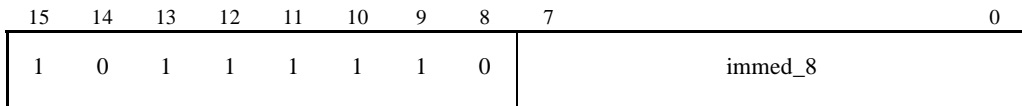
15	14	13	12	11	10	9	8	7		0
1	0	1	1	1	1	1	0		immed_8	

The BKPT (Breakpoint) instruction 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 the BKPT instruction 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, as an FIQ interrupt can occur within an abort handler.

*   Debug hardware overrides 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

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 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 will occur.

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]

7.1.17   BL, BLX(1)

15	14	13	12	11	10		0
1	1	1	H			offset_11	
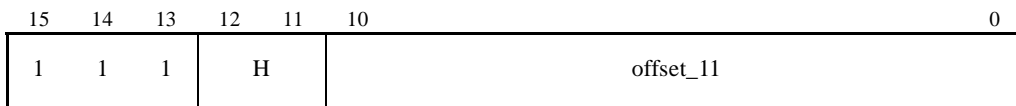
This form of the `BL` (Branch with Link) instruction provides an unconditional subroutine call to another Thumb routine. The return from subroutine is typically performed by one of the following:

- `MOV PC,LR`
- `BX LR`
- a `POP` instruction that loads the PC.

This form of the `BLX` (Branch with Link and Exchange instruction set) instruction 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.

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
    T Flag = 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:

$$-2^{22} <= \text{offset} <= +2^{22} - 2$$

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 A7-20).
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.

 ARM DDI 0100E

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]
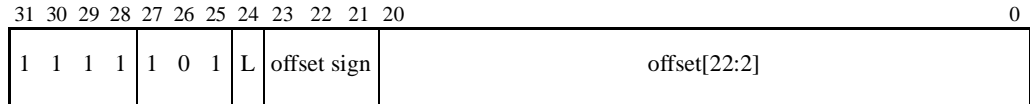
where L == offset[1].

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]
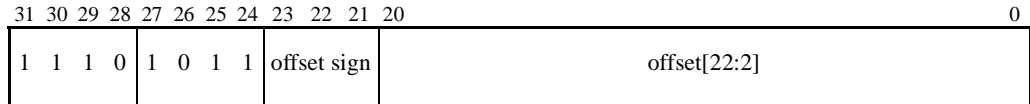
These differ slightly from the Thumb instruction pairs because of the different values of the PC in ARM and Thumb code. This can be compensated for by adjusting the offset by 4.

7.1.18 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		

This form of BLX is used to call 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 T flag 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
T Flag = 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

7.1.19   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		

The BX (Branch and Exchange) instruction is used to branch 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

```
T Flag = 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.

7.1.20 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	

The CMN (Compare Negative) instruction 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	

7.1.21   CMP (1)

15	14	13	12	11	10		8	7		0
0	0	1	0	1		Rn			immed_8	
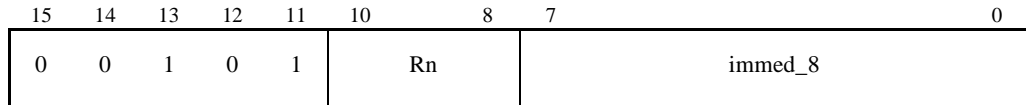
This form of the CMP (Compare) instruction compares a register value with a large immediate value. The condition flags are updated, based on the result of subtracting the constant from the register value, so that subsequent instructions can be conditionally executed (using a conditional branch).

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	
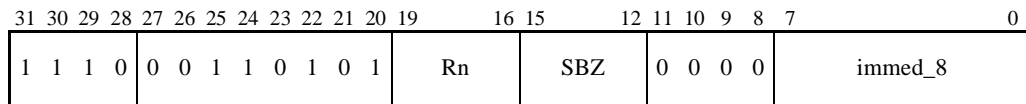
7.1.22 CMP (2)

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	

This form of CMP 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		

7.1.23 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		

This form of CMP 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.

Equivalent ARM syntax and encoding

A close equivalent is:

```
CMP   <Rn>, <Rm>
```
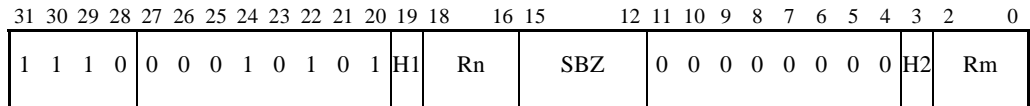
31 30 29 28	27 26 25 24 23 22 21 20	19	18 16	15 12	11 10 9 8 7 6 5 4	3	2 0
1 1 1 0	0 0 0 1 0 1 0 1	H1	Rn	SBZ	0 0 0 0 0 0 0 0	H2	Rm

There are slight differences when the instruction accesses the PC, because of the different definitions of the PC when executing ARM and Thumb code.

7.1.24   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	

The EOR (Exclusive OR) instruction performs a bitwise EOR of the values in two registers. The condition code flags are updated, 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		

7.1.25   LDMIA

15	14	13	12	11	10	8	7	0
1	1	0	0	1	Rn		register_list	
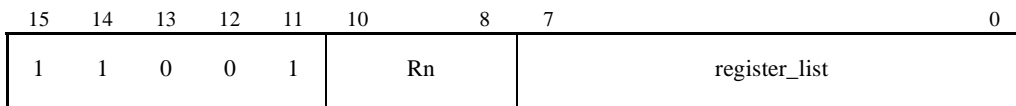
The LDMIA (Load Multiple Increment After) instruction loads a non-empty subset, or possibly all, of the general-purpose registers from sequential memory locations.

Syntax

LDMIA   <Rn>!, <registers>

where:

<Rn>   Is the register containing the start address for the instruction.

!   Causes base register writeback, 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

The LDMIA instruction is useful 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 A2-17.

Alignment   Load Multiple instructions ignore the least significant two bits of address.

If an implementation includes a System Control coprocessor (see Chapter B2 *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 A2-32 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 © 1996-2000 ARM Limited. All rights reserved.

7.1.26 LDR (1)

15	14	13	12	11	10	6	5	3	2	0
0	1	1	0	1	immed_5		Rn		Rd	
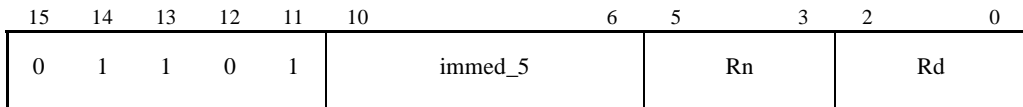
This form of the LDR (Load Register) instruction allows 32-bit memory data to be loaded into a general-purpose register. The addressing mode is useful for accessing structure (record) fields. With an offset of zero, the address produced is the unaltered value of the base register <Rn>.

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
```

Notes

Data abort   For details of the effects of the instruction if a data abort occurs, see *Effects of data-aborted instructions* on page A2-17.

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 B2 *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

7.1.27 LDR (2)
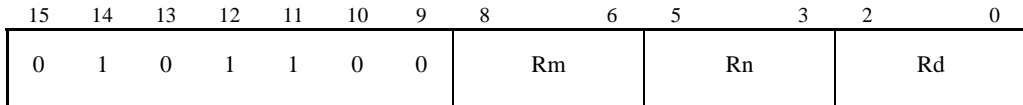
15	14	13	12	11	10	9	8		6	5		3	2		0
0	1	0	1	1	0	0		Rm			Rn			Rd	

This form of LDR allows 32-bit memory data to be loaded 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 A2-17.

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 B2 *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

7.1.28 LDR (3)
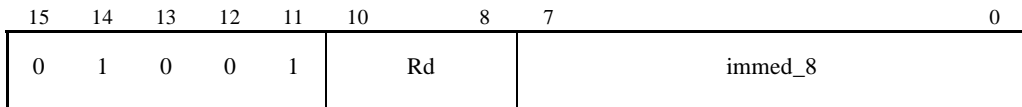
15	14	13	12	11	10 8	7 0
0	1	0	0	1	Rd	immed_8

This form of LDR allows 32-bit memory data to be loaded 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 A2-17.

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.

7.1.29   LDR (4)
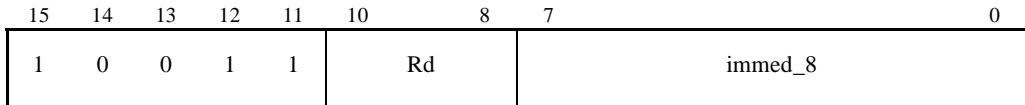
15	14	13	12	11	10	8	7	0
1	0	0	1	1		Rd		immed_8

This form of LDR allows 32-bit memory data to be loaded 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 A2-17.

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 B2 *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

7.1.30   LDRB (1)

15	14	13	12	11	10	6	5	3	2	0
0	1	1	1	1	immed_5		Rn		Rd	

This form of the LDRB (Load Register Byte) instruction loads a byte from memory and zero-extends it to form a 32-bit word which is written 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 A2-17.

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

7.1.31   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	

This form of LDRB loads a byte from memory and zero-extends it to form a 32-bit word, which is written 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 A2-17.

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	

7.1.32   LDRH (1)

15	14	13	12	11	10	6	5	3	2	0
1	0	0	0	1	immed_5		Rn		Rd	
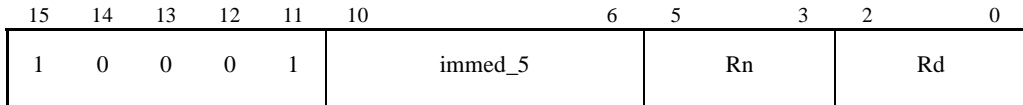
This form of the LDRH (Load Register Halfword) instruction loads a halfword (16 bits) from memory and zero-extends it to form a 32-bit word, which is written to a general-purpose register. The addressing mode is useful for accessing structure (record) fields. With an offset of zero, the address produced 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 A2-17.

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 B2 *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 © 1996-2000 ARM Limited. All rights reserved.

7.1.33   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	

This form of LDRH loads a halfword (16 bits) from memory and zero-extends it to form a 32-bit word, which is written 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 A2-17.

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 B2 *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

7.1.34   LDRSB

15	14	13	12	11	10	9	8		6	5		3	2		0
0	1	0	1	0	1	1		Rm			Rn			Rd	

The LDRSB (Load Register Signed Byte) instruction loads a byte from memory and sign-extends it to form a 32-bit word, which is written 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 A2-17.

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		

7.1.35 LDRSH

15	14	13	12	11	10	9	8	6	5	3	2	0
0	1	0	1	1	1	1	Rm		Rn		Rd	

The LDRSH (Load Register Signed Halfword) instruction loads a halfword from memory and sign-extends it to form a 32-bit word, which is written 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 A2-17.

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 B2 *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

 ARM DDI 0100E
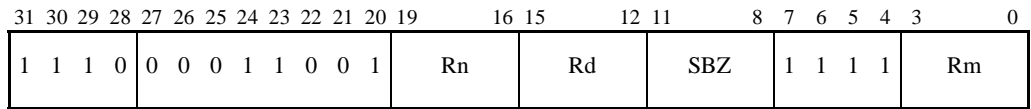
7.1.36 **LSL (1)**

15	14	13	12	11	10					6	5		3	2		0
0	0	0	0	0			immed_5					Rm			Rd	

This form of the LSL (Logical Shift Left) instruction is used to provide either the value of a register directly (LSL #0), or the value of a register multiplied by a constant power of two. Zeros are inserted into the bit positions vacated by the shift, and the condition code flags are updated, 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
```

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

7.1.37 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		

This form of LSL is used to provide the value of a register multiplied by a variable power of two. Zeros are inserted into the vacated bit positions. The condition code flags are updated, 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

7.1.38   LSR (1)

15	14	13	12	11	10						6	5			3	2			0
0	0	0	0	1	immed_5							Rm				Rd			
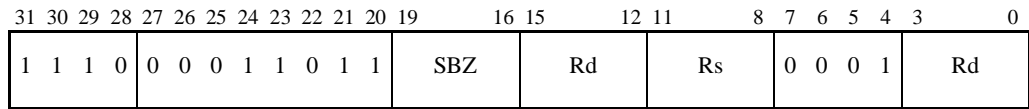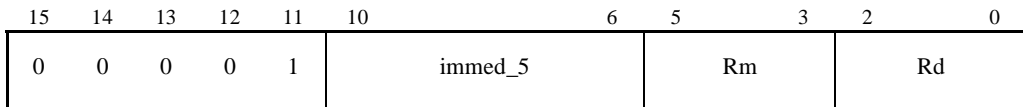
This form of the LSR (Logical Shift Right) instruction is used to provide the unsigned value of a register, divided by a constant power of two. LSR performs a logical shift right of the value of register <Rm>, and zeros are inserted into the vacated bit positions. The condition code flags are updated, 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

7.1.39 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	

This form of LSR is used to provide the unsigned value of a register divided by a variable power of two. Zeros are inserted into the vacated bit positions. The condition code flags are updated, 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

Copyright © 1996-2000 ARM Limited. All rights reserved.

7.1.40 MOV (1)

15	14	13	12	11	10		8	7		0
0	0	1	0	0		Rd			immed_8	
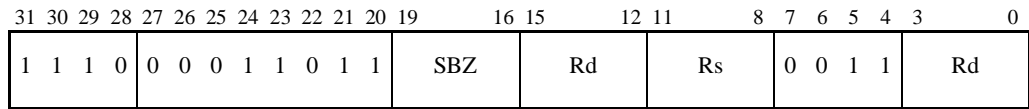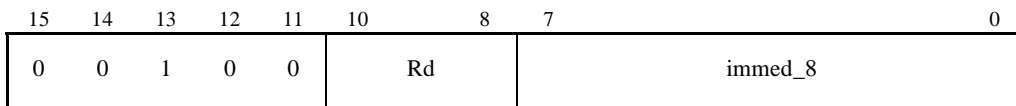
This form of the MOV (Move) instruction moves a large immediate value to a register. The condition code flags are updated, based on the result.

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	
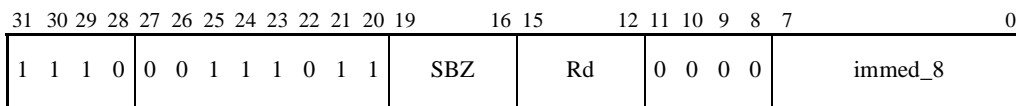
7.1.41 MOV (2)

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	

This form of MOV is used to move a value from one low register to another, and the flags are set according to that 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.

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

7.1.42   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		

This form of MOV is used to move a value to, from, or between high registers. Unlike the low register MOV instruction described in *MOV (2)* on page A7-67, 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

Operand restriction   If a low register is specified for <Rd> and <Rm> (H1==0 and H2==0), the result is UNPREDICTABLE.

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.

7.1.43 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	

The MUL (Multiply) instruction multiplies signed or unsigned variables to produce a 32-bit result. The condition code flags are updated, 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
```

Notes

Operand restriction   Specifying the same register for `<Rd>` and `<Rm>` has UNPREDICTABLE results.

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 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 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.

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 A4-66) and might have the wrong early termination behavior:

```
MULS   <Rd>, <Rd>, <Rm>
```

7.1.44 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	

The MVN (Move NOT) instruction is used to complement a register value, often to form a bit mask. The condition code flags are updated, 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		

7.1.45 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		

The NEG (Negate) instruction negates the value of one register and stores the result in a second register. The condition code flags are updated (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

7.1.46   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	

The ORR (Logical OR) instruction performs a bitwise OR of the values in two registers. The condition code flags are updated, 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		

7.1.47 POP

15	14	13	12	11	10	9	8	7	0
1	0	1	1	1	1	0	R	register_list	
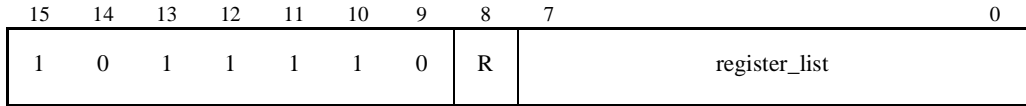
The POP (Pop Multiple Registers) instruction loads a subset (or possibly all) of the general-purpose registers R0-R7 and the PC from the stack.

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

The POP instruction is useful 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 A2-17.

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 B2 *The System Control Coprocessor*) and alignment checking is enabled, an address with bits[1:0] != 0b00 causes an alignment exception.

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 A2-32 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

Copyright © 1996-2000 ARM Limited. All rights reserved.

7.1.48   **PUSH**

15	14	13	12	11	10	9	8	7	0
1	0	1	1	0	1	0	R	register_list	
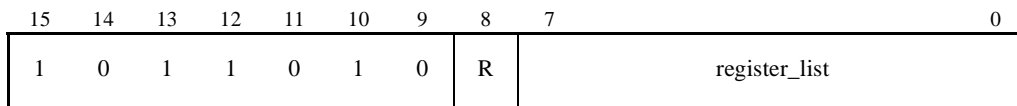
The PUSH (Push Multiple Registers) instruction stores a subset (or possibly all) of the general-purpose registers R0-R7 and the LR to the stack.

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

PUSH is useful 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 A2-17.

Alignment   PUSH instructions ignore the least significant two bits of address.

If an implementation includes a System Control coprocessor (see Chapter B2 *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 A2-32 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

7.1.49   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	

The ROR (Rotate Right Register) instruction is used to provide the value of a register rotated by a variable value (from another register). The bits that are rotated off the right end are inserted into the vacated bit positions on the left. The condition code flags are updated, 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

Copyright © 1996-2000 ARM Limited. All rights reserved.
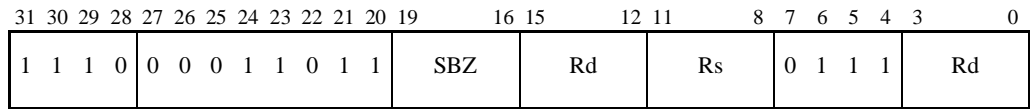
7.1.50   SBC

15	14	13	12	11	10	9	9	8	6	5		3	2		0
0	1	0	0	0	0	0	1	1	0		Rm			Rd	

The SBC (Subtract with Carry) instruction can be used to synthesize multi-word subtraction. It subtracts the value of register <Rm> and the value of NOT(Carry Flag) from the value of register <Rd>. The condition code flags are updated, based on the result.

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

7.1.51 STMIA

15	14	13	12	11	10	8	7	0
1	1	0	0	0	Rn		register_list	
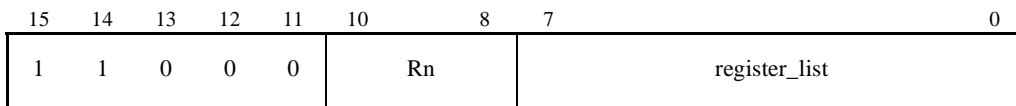
The STMIA (Store Multiple Increment After) instruction stores a non-empty subset, or possibly all, of the general-purpose registers to sequential memory locations.

Syntax

STMIA   <Rn>!, <registers>

where:

<Rn>	Is the register containing the start address for the instruction.
!	Causes base register writeback, 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 writeback 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 A2-17.

Alignment   Store Multiple instructions ignore the least significant two bits of address.

If an implementation includes a System Control coprocessor (see Chapter B2 *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 A2-32 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

7.1.52 STR (1)

15	14	13	12	11	10						6	5		3	2		0
0	1	1	0	0			immed_5						Rn			Rd	
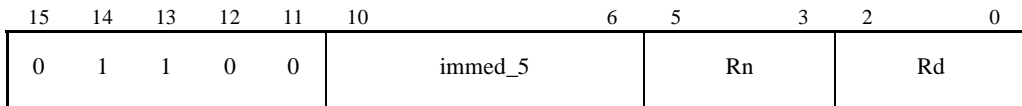
This form of the STR (Store Register) instruction allows 32-bit data from a general-purpose register to be stored to memory. The addressing mode is useful for accessing structure (record) fields. With an offset of zero, the address produced is the unaltered value of the base register <Rn>.

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 A2-17.

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 B2 *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 0	immed_5	0 0

7.1.53 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	

This form of STR allows 32-bit data from a general-purpose register to be stored 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 A2-17.

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 B2 *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

7.1.54 STR (3)

15	14	13	12	11	10 8	7 0
1	0	0	1	0	Rd	immed_8

This form of STR allows 32-bit data from a general-purpose register to be stored 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 A2-17.

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 B2 *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

7.1.55   STRB (1)

15	14	13	12	11	10	6	5	3	2	0
0	1	1	1	0	immed_5		Rn		Rd	

This form of the STRB (Store Register Byte) instruction allows 8-bit data from a general-purpose register to be stored 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 A2-17.

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	

7.1.56   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	

This form of STRB allows 8-bit data from a general-purpose register to be stored 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 A2-17.

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	

7.1.57 STRH (1)

15	14	13	12	11	10 6	5 3	2 0
1	0	0	0	0	immed_5	Rn	Rd

This form of the STRH (Store Register Halfword) instruction allows 16-bit data from a general-purpose register to be stored 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
```

 ARM DDI 0100E

Notes

Data abort   For details of the effects of the instruction if a data abort occurs, see *Effects of data-aborted instructions* on page A2-17.

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 B2 *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

7.1.58   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	

This form of STRH allows 16-bit data from a general-purpose register to be stored 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 A2-17.

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 B2 *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

7.1.59 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		

This form of the SUB (Subtract) instruction subtracts a small constant value from the value of a register and stores the result in a second register. The condition code flags are updated, 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

7.1.60 SUB (2)

15	14	13	12	11	10 8	7 0
0	0	1	1	1	Rd	immed_8

This form of SUB subtracts a large immediate value from 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

```
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
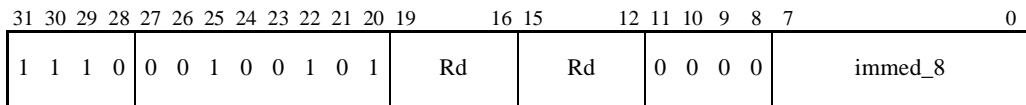
7.1.61   SUB (3)

15	14	13	12	11	10	9	8		6	5		3	2		0
0	0	0	1	1	0	1		Rm			Rn			Rd	

This form of SUB subtracts the value of one register from the value of a second register and stores the result in a third register. The condition code flags are updated, 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	

7.1.62   SUB (4)

15	14	13	12	11	10	9	8	7	6			0
1	0	1	1	0	0	0	0	1		immed_7		
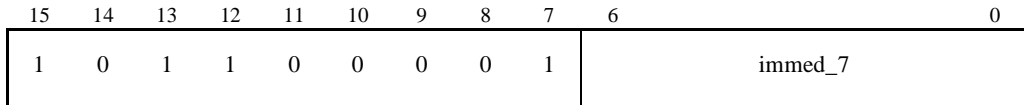
This form of SUB decrements the SP by four times a 7-bit immediate (that is, by a multiple of 4 in the range 0 to 508). The condition codes are not affected.

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
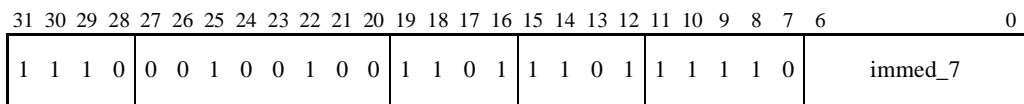
7.1.63   SWI

15	14	13	12	11	10	9	8	7								0
1	1	0	1	1	1	1	1	immed_8								

The SWI (Software Interrupt) instruction is used as an operating system service call. A SWI exception is generated, which is handled by an operating system to provide the requested service. See *Exceptions* on page A2-13.

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							

7.1.64 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	

The TST (Test) instruction is used to determine 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. The condition code flags are updated, 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		

7.2 Thumb instructions and architecture versions

Table 7-1 shows which Thumb instructions are present in each current ARM architecture version that supports Thumb.

Table 7-1 Thumb instructions by architecture

Instruction	v4T, v4TxM	v5T, v5TxM
ADC	Yes	Yes
ADD (all forms)	Yes	Yes
AND	Yes	Yes
ASR (both forms)	Yes	Yes
B (both forms)	Yes	Yes
BIC	Yes	Yes
BKPT	No	Yes
BL	Yes	Yes
BLX (both forms)	No	Yes
BX	Yes	Yes
CMN	Yes	Yes
CMP (all forms)	Yes	Yes
EOR	Yes	Yes
LDMIA	Yes	Yes
LDR (all forms)	Yes	Yes
LDRB (both forms)	Yes	Yes
LDRH (both forms)	Yes	Yes
LDRSB	Yes	Yes
LDRSH	Yes	Yes
LSL (both forms)	Yes	Yes
LSR (both forms)	Yes	Yes
MOV (all forms)	Yes	Yes

Table 7-1 Thumb instructions by architecture *(Continued)*

Instruction	v4T, v4TxM	v5T, v5TxM
MUL	Yes	Yes
MVN	Yes	Yes
NEG	Yes	Yes
ORR	Yes	Yes
POP	Yes	Yes
PUSH	Yes	Yes
ROR	Yes	Yes
SBC	Yes	Yes
STMIA	Yes	Yes
STR (all forms)	Yes	Yes
STRB (both forms)	Yes	Yes
STRH (both forms)	Yes	Yes
SUB (all forms)	Yes	Yes
SWI	Yes	Yes
TST	Yes	Yes

Chapter A8
The 26-bit Architectures

This chapter describes the differences between the 32-bit and 26-bit architectures. It contains the following sections:

- *Overview of the 26-bit architectures* on page A8-2
- *Format of register 15* on page A8-4
- *26-bit PSR update instructions* on page A8-6
- *Address exceptions* on page A8-8
- *Backwards compatibility from 32-bit architectures* on page A8-9.

——— **Note** ———

The information in this chapter is to help with *backwards-compatibility* issues in existing systems. Programmers starting new projects are strongly encouraged to use only the 32-bit facilities.

8.1 Overview of the 26-bit architectures

ARMv1, ARMv2, and ARMv2a are earlier versions of the ARM architecture which implement only a 26-bit address space, and are known as 26-bit architectures.

ARM architecture version 3 and above implement a 32-bit address space and are known as 32-bit architectures. For backwards compatibility:

- except for ARMv3G, all variants of ARM architecture version 3 implement the 26-bit address space

- all non-T variants of ARM architecture version 4 and above can optionally implement the 26-bit address space.

This chapter describes the differences between the 26-bit and 32-bit architectures, and how the backwards-compatibility features are used.

There are several differences between the 26-bit and the 32-bit architectures:

Program counter The 26-bit architectures implement only a 24-bit program counter in R15, which allows 64MB of program space. The 32-bit architectures have a 30-bit program counter in R15 which allows 4GB of program space on 32-bit architectures.

Processor modes Only four processor modes are supported on 26-bit architectures:
- User (0b00)
- FIQ (0b01)
- IRQ (0b10)
- Supervisor (0b11).

Register 15 In the 26-bit architectures, the following are also stored in register 15:
- four condition flags (N, Z, C and V)
- the interrupt disable flags (I and F)
- two processor mode bits (M1 and M0).

CPSR/SPSR The 26-bit architectures do not have a CPSR or any SPSRs.

Exceptions An exception (called an address exception) is raised if a memory access instruction uses an address that is greater than $2^{26}-1$ bytes.

Branches In 26-bit architectures, there are no restrictions on branching backwards past location 0x0000000 or forwards past location 0x3FFFFFF. Such branches wrap around to the other end of the 26-bit address space, and so have a different target address than they would have had in a 32-bit architecture.

As a result, the signed 24-bit word offset in the B and BL instructions allows any instruction in the 26-bit address space to be branched to.

Together, these differences make up the fundamental distinction between 26-bit and 32-bit architectures:

26-bit architectures

> All process status (namely the condition flags, interrupt status and processor mode) can be preserved across subroutine calls and nested exceptions without adding any instructions to the entry or exit sequence.

32-bit architectures

> This process status functionality is given up to allow 32-bit instruction addresses to be used. For exceptions, processor status is preserved in the SPSRs, and if nested exceptions using the same SPSR can occur, extra instructions are used to preserve this status in memory. For subroutine calls, processor status can be preserved across the subroutine call by using extra instructions, but this is not normally done.

8.2 Format of register 15

31	30	29	28	27	26	25			2	1	0
N	Z	C	V	I	F		Program Counter			M1	M0
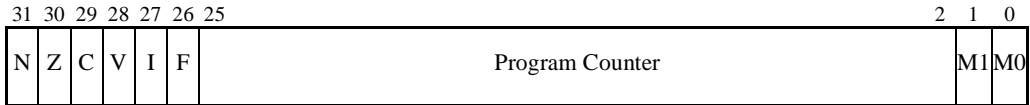
Bits[25:2] are collectively known as the *Program Counter* (PC). Because the PC occupies only 24 bits of register 15, only 2^{24} instructions (2^{26} bytes) can be addressed, giving a maximum addressable program size of 64MB.

Bits[31:26] and bits[1:0] are collectively known as the *Program Status Register* (PSR).

The N, Z, C, V, I, and F bits have the same meaning in both 26-bit and 32-bit architectures. M[1:0] also have the same meaning in both architectures.

Abort, Undefined, and System mode are not supported in 26-bit architectures. Aborts and Undefined Instruction exceptions have exactly the same actions in both modes, except that in 26-bit architectures, Supervisor mode is entered instead of Abort or Undefined mode.

The precise effect of an exception on a 26-bit architecture can be derived from the rules in *Exceptions* on page A2-13, as follows:

•   The banked version of R14 has bits[25:2] set to the specified address, and bits[31:26, 1, 0] set to copies of the corresponding bits in R15.

•   The I, F, M1, and M0 bits are modified in the same way as CPSR[7], CPSR[6], CPSR[1], and CPSR[0] respectively, on a 32-bit architecture.

The I, F and M[1:0] bits cannot be written directly when the processor is in User mode. In User mode they are only changed by an exception occurring.

8.2.1 Reading register 15

In 26-bit architectures, the value of R15 is read in five different ways:

•   Most importantly, if R15 has an UNPREDICTABLE value in the 32-bit architecture, it also has an UNPREDICTABLE value when used in the same way in the 26-bit architecture.

•   If R15 is specified in bits[19:16] of an instruction (and its value is not UNPREDICTABLE), only the PC (bits[25:2]) is used. All other bits read as zero.

•   If R15 is specified in bits[3:0] of an instruction (and its value is not UNPREDICTABLE), all 32 bits are used.

•   If R15 is stored using STR or STM, the value of the PC (bits[25:2]) is IMPLEMENTATION DEFINED, but all 32 bits of the register are stored.

•   All 32 bits are stored in the Link register (R14) after a Branch with Link instruction or an exception entry.

8.2.2 Writing register 15

In 26-bit architectures, the value of R15 is written in three different ways:

- The following instructions only write the PC part of R15, leaving the PSR part unchanged:
  - — Data-processing instructions without the S bit set
  - — LDR instructions
  - — LDM instructions, other than Load Multiple with Restore CPSR.

- The following instructions write both the PC and the PSR part of R15:
  - — Data-processing instructions with the S bit set
  - — Load Multiple with Restore CPSR.

- Variants of the CMP, CMN, TST, and TEQ instructions write just the PSR part of R15 and leave the PC part unchanged. These instruction variants are described *26-bit PSR update instructions* on page A8-6.

These read/write rules mean that R15 is used in three basic ways:

- When it is the Rn specifier in data-processing instructions, or the base address for load and store instructions, only the value of the program counter is used, to simplify PC-relative addressing and position-independent code.

- When it is the Rm specifier in data-processing instructions, all 32 bits are used in order to allow all process status to be restored after a subroutine call or exception by subroutine-return instructions such as:

  ```
  MOVS PC, LR and LDM..., PC}^
  ```

  These instructions are UNPREDICTABLE in User mode on 32-bit architectures, but are legal on 26-bit architectures, as they are used to restore the condition code flags on procedure exit.

- All 32 bits are saved in the Link register, to preserve the PC and the PSR across subroutine calls and exceptions.

8.3   26-bit PSR update instructions

31	27 26	25	24 23	22 21	20	19	16	15	12	11	0
cond	0 0	I	1 0	opc	1	Rn		1 1 1 1		shifter_operand	
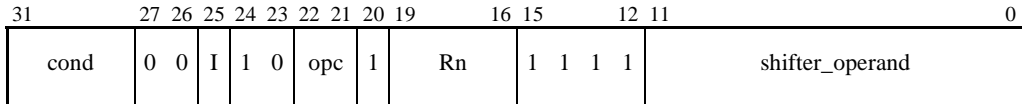
In 26-bit architectures, the MSR and MRS instructions are not supported. Instead, variants of the CMN, CMP, TEQ and TST instructions are used to write just the PSR part of R15 without affecting the PC part. These 26-bit-only variants are called CMNP, CMPP, TEQP and TSTP, and are distinguished by having instruction bits[15:12] equal to 0b1111. These bits are equal to 0b0000 in the normal variants of these instructions.

These instructions write their ALU result directly to the PSR part of register 15 (only N, Z, C, and V are affected in User mode).

When the processor is in User mode, only the condition codes are affected. All other modes allow all PSR bits to be altered.

8.3.1   Syntax

```
CMN{<cond>}P  Rn, <shifter_operand>
CMP{<cond>}P  Rn, <shifter_operand>
TEQ{<cond>}P  Rn, <shifter_operand>
TST{<cond>}P  Rn, <shifter_operand>
```

where:

<cond>   Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-5. If <cond> is omitted, the AL (always) condition is used.

<Rn>   Specifies the register that contains the first operand for the instruction.

<shifter_operand>

Specifies the second operand for the instruction. Its options 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.

8.3.2   Exceptions

None

 ARM DDI 0100E

8.3.3   Operation

```
if ConditionPassed(<cond>) then
    case <opc> of
        0b00 /* TSTP */
            <alu_out> = Rn AND <shifter_operand>
        0b01 /* TEQP */
            <alu_out> = Rn EOR <shifter_operand>
        0b10 /* CMPP */
            <alu_out> = Rn - <shifter_operand>
        0b11 /* CMNP */
            <alu_out> = Rn + <shifter_operand>
    endcase
    if R15[1:0] == 0b00 then                /* M[1:0] == 0b00, User mode */
        R15[31:28] = <alu_out>[31:28]       /* update just NZCV */
    else                                    /* a privileged mode */
        R15[31:26] = <alu_out>[31:26]       /* update NZCVIF and ... */
        R15[1:0] = <alu_out>[1:0]           /* ... update M[1:0] */
```

8.4   Address exceptions

In 26-bit architectures, all data addresses are checked to ensure that they are between 0 and 64MB. If a data address is produced with a 1 in any of the top 6 bits, an address exception is generated. When an address exception is generated, the following actions are performed:

```
R14_svc[25:2]           =       address of instruction + 8
R14_svc[31:26,1,0]      =       R15[31:26,1,0]
M[1:0] =        0b11                ; Supervisor mode
F      =        unchanged
I      =        1                   ; (normal) interrupts disabled
PC     =        0x14
```

The address of the instruction which caused the address exception is the value in R14 minus 8.

8.4.1   Returning from an address exception

As this exception implies a programming error, it is not usual to return from address exceptions, but if a return is required, use:

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

This restores both the PC and PSR (from R14_svc) and returns to the instruction that generated the address exception.

                   ARM DDI 0100E

8.5   Backwards compatibility from 32-bit architectures

As described in *Overview of the 26-bit architectures* on page A8-2, some 32-bit architectures include mandatory or optional backwards-compatibility support for 26-bit architectures. Architecture versions with this support implement the four 26-bit modes described in *Overview of the 26-bit architectures* on page A8-2. When the processor is in one of these modes, the PC behaves as explained in *Format of register 15* on page A8-4 and the instructions described in *26-bit PSR update instructions* on page A8-6 are available. This allows older 26-bit programs to be executed in a suitable 26-bit mode.

In architecture versions without 26-bit backwards-compatibility support, CPSR bit[4] (M[4]) always reads as 1, and all writes to it are ignored.

The complete list of processor modes is shown in Table 8-1.

Table 8-1 32-bit and 26-bit modes

M[4:0]	Mode	Accessible registers
0b00000	User_26	R0 to R14, PC, (CPSR)
0b00001	FIQ_26	R0 to R7, R8_fiq to R14_fiq, PC, (CPSR, SPSR_fiq)
0b00010	IRQ_26	R0 to R12, R13_irq, R14_irq, PC, (CPSR, SPSR_irq)
0b00011	SVC_26	R0 to R12, R13_svc, R14_svc, PC, (CPSR, SPSR_svc)
0b10000	User_32	R0 to R14, PC, CPSR
0b10001	FIQ_32	R0 to R7, R8_fiq to R14_fiq, PC, CPSR, SPSR_fiq
0b10010	IRQ_32	R0 to R12, R13_irq, R14_irq, PC, CPSR, SPSR_irq
0b10011	SVC_32	R0 to R12, R13_svc, R14_svc, PC, CPSR, SPSR_svc
0b10111	Abort_32	R0 to R12, R13_abt, R14_abt, PC, CPSR, SPSR_abt
0b11011	Undef_32	R0 to R12, R13_und, R14_und, PC, CPSR, SPSR_und
0b11111	System_32	R0 to R14, PC, CPSR (architecture version 4 and above)

─────── **Note** ───────

The MRS and MSR instructions are supported in all cases, despite the fact that they do not exist in pure 26-bit architectures. This is to assist conversion to a 32-bit architecture, as the CMNP, CMPP, TEQP, and TSTP instructions cannot enter a 32-bit mode from a 26-bit mode.

If an MSR instruction which lies outside the 26-bit address space is used to switch into a 26-bit mode, the results are UNPREDICTABLE.

The results of an MSR instruction which switches between a 32-bit mode and a 26-bit mode (in either direction) are also UNPREDICTABLE if it is executed at any point where predictable subsequent execution can reach either of the following:

- the end of the 26-bit address space
- a branch instruction whose target addresses in the 26-bit and 32-bit address spaces are different.

For this purpose, *predictable subsequent execution* is as defined in *Other uses for IMBs* on page A2-30.

───────────────────

8.5.1 32-bit and 26-bit configuration

Architecture versions with 26-bit backwards compatibility support optionally incorporate two hardware signals that control 32-bit instruction accesses and 32-bit data accesses. The signals are mapped to two bits in register 1 of the system control coprocessor. These signals are:

- **PROG32**
- **DATA32**.

─────── **Note** ───────

If an implementation does not support these signals but does provide 32-bit modes, it must behave as though both signals were 1.

───────────────────

32-bit configuration

1. If **PROG32** is active and **DATA32** is active, the processor switches to a 32-bit mode when processing exceptions (including Reset), using the _32 modes for handling all exceptions. This is called a 32-bit configuration. Abort_32 mode is used for handling memory aborts, and Undef_32 for handling Undefined Instruction exceptions. A 26-bit mode can be selected by putting a 26-bit mode number into the M[4:0] bits of the CPSR (either using MSR or an exception return sequence). A 32-bit mode can also be entered from a 26-bit mode using the MSR instruction. Once in a 26-bit mode, another 26-bit mode can be entered using one of the CMNP, CMPP, TEQP and TSTP instructions, or the MSR instruction.

   If an exception occurs when the processor is in a 26-bit mode, only the PC bits from R15[25:2] are copied to the link register. The remaining bits in the link register are zeroed. The PSR bits from R15[31:26] and R15[1:0] are copied into the SPSR, ready for a normal 32-bit return sequence.

2. If **PROG32** is active and **DATA32** is not active (32-bit programs with 26-bit data), the result is UNPREDICTABLE.

26-bit configuration

1. If **PROG32** is not active, the processor is locked into 26-bit modes (that is, cannot be placed into a 32-bit mode by any means) and handles exceptions in 26-bit modes. This is called a 26-bit configuration. In this configuration, CMNP, CMPP, TEQP and TSTP instructions, or the MSR instruction can be used to switch to 26-bit modes. Attempts to write CPSR bits[4:2] (M[4:2]) are ignored, stopping any attempts to switch to a 32-bit mode, and SVC_26 mode is used to handle memory aborts and Undefined Instruction exceptions. The PC is limited to 24 bits, limiting the addressable program memory to 64MB.

2. If **PROG32** is not active, **DATA32** has the following actions:
   - If **DATA32** is not active, all data addresses are checked to ensure that they are between 0 and 64MB. If a data address is produced with a 1 in any of the top 6 bits, an address exception is generated.
   - If **DATA32** is active, full 32-bit addresses can be produced and are not checked for address exceptions. This allows 26-bit programs to access data in the full 32-bit address space.

8.5.2 Vector exceptions

When the processor is in a 32-bit configuration (**PROG32** is active) and in a 26-bit mode (CPSR[4] == 0), data access (but not instruction fetches) to the exception vectors (address 0x0 to 0x1F) causes a data abort. This is known as a vector exception.

Vector exceptions are always produced if the exception vectors are written in a 32-bit configuration and a 26-bit mode. It is IMPLEMENTATION DEFINED whether reading the exception vectors in a 32-bit configuration and a 26-bit mode also causes a vector exception.

Vector exceptions are provided to support 26-bit backwards compatibility. When a vector exception is generated, it indicates that a 26-bit mode process is trying to install a (26-bit) vector handler. Because the processor is in a 32-bit configuration, exceptions are handled in a 32-bit mode, so a veneer must be used to change from the 32-bit exception mode to a 26-bit mode before calling the 26-bit exception handler. This veneer can be installed on each vector and can switch to a 26-bit mode before calling any 26-bit handlers.

The return from the 26-bit exception handler might also need to be veneered. Some SWI handlers return status information in the processor flags, and this information needs to be transferred from the link register to the SPSR with a return veneer for the SWI handler.

Chapter A9
ARM Code Sequences

The ARM instruction set is a powerful tool for generating high-performance microprocessor systems. Used to its full extent, the ARM instruction set allows algorithms to be coded in a very compact and efficient way. This chapter describes some sample routines that provide insight into the ARM instruction set. It contains the following sections:

9.1 Arithmetic instructions

The following subsections illustrate some ways of using ARM data-processing instructions. The examples illustrate:

- *Bit field manipulation*
- *Multiplication by constant*
- *Multi-precision arithmetic* on page A9-3
- *Swapping endianness* on page A9-4.

9.1.1 Bit field manipulation

The ARM shift and logical instructions can be used for bit field manipulation:

```
; Extract 8 bits from the top of R2 and insert them into
; the bottom of R3, shifting up the data in R3
; R0 is a temporary value
    MOV    R0, R2, LSR #24         ; extract top bits from R2 into R0
    ORR    R3, R0, R3, LSL #8      ; shift up R3 and insert R0
```

9.1.2 Multiplication by constant

Combinations of shifts, add with shifts, and reverse subtract with shift can be used to perform multiplications by constants:

```
; multiplication of R0 by 2^n
    MOV    R0, R0, LSL #n           ; R0 = R0 << n

; multiplication of R0 by 2^n + 1
    ADD    R0, R0, R0, LSL #n       ; R0 = R0 + (R0 << n)

; multiplication of R0 by 2^n - 1
    RSB    R0, R0, R0, LSL #n       ; R0 = (R0 << n) - R0

; R0 = R0 * 10 + R1
    ADD    R0, R0, R0, LSL #2       ; R0 = R0 * 5
    ADD    R0, R1, R0, LSL #1       ; R0 = R1 + R0 * 2

; R0 = R0 * 100 + R1
    ADD    R0, R0, R0, LSL #2       ; R0 = R0 * 5
    ADD    R0, R0, R0, LSL #2       ; R0 = R0 * 5 (R0 = R0 * 25)
    ADD    R0, R1, R0, LSL #2       ; R0 = R1 + R0 * 4
```

9.1.3 Multi-precision arithmetic

Arithmetic instructions allow efficient arithmetic on 64-bit or larger objects:

- Add, and Add with Carry perform multi-precision addition
- Subtract, and Subtract with Carry perform subtraction
- Compare can be used for comparison.

```
; On entry      :   R0 and R1 hold a 64-bit number
;                :   (R0 is least significant)
;                :   R2 and R3 hold a second 64-bit number
; On exit       :   R0 and R1 hold 64-bit sum (or difference) of the 2 numbers

add64    ADDS    R0, R0, R2        ; add lower halves and update Carry flag
         ADC     R1, R1, R3        ; add the high halves and Carry flag

sub64    SUBS    R0, R0, R2        ; subtract lower halves, update Carry
         SBC     R1, R1, R3        ; subtract high halves and Carry

; This routine compares two 64-bit numbers
; On entry      : As above
; On exit       : N, Z, and C flags updated correctly
cmp64    CMP     R1, R3            ; compare high halves, if they are
         CMPEQ   R0, R2            ; equal, then compare lower halves
```

Be aware that in the above example, the V flag is not updated correctly. For example:

```
    R1 = 0x00000001,   R0 = 0x80000000
    R3 = 0x00000001,   R2 = 0x7FFFFFFF
```

R0 − R2 overflows as a 32-bit signed number, so the CMPEQ instruction sets the V flag. But (R1, R0) − (R3, R2) does not overflow as a 64-bit number.

An alternative routine exists which updates the V flag correctly, but not the Z flag:

```
; This routine compares two 64-bit numbers
; On entry: as above
; On exit: N, V and C set correctly            ; R4 is destroyed
cmp64    SUBS    R4, R0, R2
         SBCS    R4, R1, R3
```

9.1.4   Swapping endianness

Swapping the order of bytes in a word (the endianness) can be performed in two ways:

- This method is best for single words:

```
; On entry     :    R0 holds the word to be swapped
; On exit      :    R0 holds the swapped word, R1 is destroyed

byteswap                          ; R0 = A , B , C , D
    EOR    R1, R0, R0, ROR #16    ; R1 = A^C,B^D,C^A,D^B
    BIC    R1, R1, #0xFF0000      ; R1 = A^C, 0 ,C^A,D^B
    MOV    R0, R0, ROR #8         ; R0 = D , A , B , C
    EOR    R0, R0, R1, LSR #8     ; R0 = D , C , B , A
```

- This method is best for swapping the endianness of a large number of words:

```
; On entry     : R0 holds the word to be swapped
; On exit      : R0 holds the swapped word,
;               : R1, R2 and R3 are destroyed

byteswap                          ; first the two-instruction initialization
    MOV    R2, #0xFF          ; R2 = 0xFF
    ORR    R2, R2, #0xFF0000  ; R2 = 0x00FF00FF

    ; repeat the following code for each word to swap
                              ; R0 = A   B   C   D
    AND    R1, R2, R0         ; R1 = 0   B   0   D
    AND    R0, R2, R0, ROR #24 ; R0 = 0   C   0   A
    ORR    R0, R0, R1, ROR #8  ; R0 = D   C   B   A
```

       ARM DDI 0100E

9.2 Branch instructions

The following subsections show some different ways of controlling the flow of execution in ARM code.

9.2.1 Procedure call and return

The `BL` (Branch and Link) instruction makes a procedure call by preserving the address of the instruction after the `BL` in R14 (the link register, LR), and then branching to the target address. Returning from a procedure is achieved by moving R14 to the PC:

```
    ....
    BL    function            ; call 'function'
    ....                       ; procedure returns to here
    ....
function                       ; function body
    ....
    ....
    MOV   PC, LR               ; Put R14 into PC to return
```

Another method to return from a called procedure is given in *Procedure entry and exit* on page A9-10.

9.2.2 Conditional execution

Conditional execution allows *if-then-else* statements to be collapsed into sequences that do not require forward branches:

```
/* C code for Euclid's Greatest Common Divisor (GCD)*/
/* Returns the GCD of its two parameters */
int gcd(int a, int b)
{       while (a != b)
            if (a > b )
                a = a - b ;
            else
                b = b - a ;
        return a ;
}

; ARM assembler code for Euclid's Greatest Common Divisor
; On entry: R0 holds 'a', R1 holds 'b'
; On exit : R0 hold GCD of A and B
gcd     CMP     R0, R1        ; compare 'a' and 'b'
        SUBGT   R0, R0, R1    ; if (a>b) a=a-b (if a==b do nothing)
        SUBLT   R1, R1, R0    ; if (b>a) b=b-a (if a==b do nothing)
        BNE     gcd           ; if (a!=b) then keep going
        MOV     PC, LR        ; return to caller
```

9.2.3   Conditional compare instructions

Compare instructions can be conditionally executed to implement more complicated expressions:

```
if (a==0 || b==1)
        c = d + e ;

CMP     R0, #0                  ; compare a with 0
CMPNE   R1, #1                  ; if a is not 0, compare b to 1
ADDEQ   R2, R3, R4              ; if either was true c = d + e
```

9.2.4   Loop variables

The Subtract instruction can be used to both decrement a loop counter and set the condition codes to test for a zero:

```
    MOV   R0, #loopcount         ; initialize the loop counter
loop                             ; loop body
    ....
    SUBS  R0, R0, #1             ; subtract 1 from counter
                                 ; and set condition codes
    BNE   loop                   ; if not zero, continue looping
    ....
```

9.2.5   Multi-way branch

A very simple multi-way branch can be implemented with a single instruction. The following code dispatches the control of execution to any number of routines, with the restriction that the code to handle each case of the multi-way branch is the same size, and that size is a power of two bytes:

```
; Multi-way branch
; On entry: R0 holds the branch index

    CMP   R0, #maxindex          ; checks the index is in range
    ADDLO PC, PC, R0, LSL #RoutineSizeLog2
                                 ; scale index by the log of the size of
                                 ; each handler, add to the PC, which points
                                 ; 2 instructions beyond this one
                                 ; (at Index0Handler), then jump there
    B     IndexOutOfRange        ; jump to the error handler

Index0Handler
    ....
    ....
Index1Handler
    ....
    ....
Index2Handler
    ....
```

9.3 Load and Store instructions

Load and Store instructions are the best way to load or store a single word. They are also the only instructions that can load or store a byte or halfword.

9.3.1 Linked lists

The following code searches for an element in a linked list that has two elements (a single byte value and a pointer to the next record) in each record. A null next pointer indicates this is the last element in the list:

```
; Linked list search
; On entry    : R0 holds a pointer to the first record in the list
;             : R1 holds the byte we are searching for
;             : Call this code with a BL
; On exit     : R0 holds the address of the first record matched
;             : or a null pointer if no match was found
;             : R2 is destroyed

llsearch
    CMP       R0, #0           ; null pointer?
    LDRNEB    R2, [R0]         ; load the byte value from this record
    CMPNE     R1, R2           ; compare with the looked-for value
    LDRNE     R0, [R0, #4]     ; if not found, follow the link to the
    BNE       llsearch         ; next record and then keep looking
    MOV       PC, LR           ; return with pointer in R0
```

9.3.2 Simple string compare

The following code performs a very simple string compare on two zero-terminated strings:

```
; String compare
; On entry    : R0 points to the first string
;             : R1 points to the second string
;             : Call this code with a BL
; On exit     : R0 is < 0 if the first string is less than the second
;             : R0 is = 0 if the first string is equal to the second
;             : R0 is > 0 if the first string is greater than the second
;             : R1, R2 and R3 are destroyed

strcmp
    LDRB      R2, [R0], #1     ; Get a byte from the first string
    LDRB      R3, [R1], #1     ; Get a byte from the second string
    CMP       R2, #0           ; Have we reached the end of either
    CMPNE     R3, #0           ;  string?
    BEQ       return           ; Go to return code if so
    CMP       R2, R3           ; Are the strings the same so far?
    BEQ       strcmp           ; Repeat for next character if so
return
    SUB       R0, R2, R3       ; Calculate result value and return
    MOV       PC, LR           ;  by copying R14 (LR) into the PC
```

The following code performs a more optimized string compare:

```
int strcmp(char *s1, char *s2)
{
    unsigned int ch1, ch2;
    do
    {
        ch1 = *s1++;
        ch2 = *s2++;
    } while (ch1 >= 1 && ch1 == ch2);
    return ch1 - ch2;
}
```

This code uses an unsigned comparison with 1 to test for a null character, rather than the normal comparison with 0.

The corresponding ARM code is:

```
strcmp
    LDRB   R2,[R0],#1
    LDRB   R3,[R1],#1
    CMP    R2,#1
    CMPCS  R2,R3
    BEQ    strcmp
    SUB    R0,R2,R3
    MOV    PC,LR
```

The change in the way that null characters are detected allows the condition tests to be combined:

- If R2 == 0, the CMP instruction sets Z = 0, C = 0. Neither the CMPCS instruction nor the BEQ instruction is executed, and the loop terminates.

- If R2 != 0 and R3 == 0, the CMP instruction sets C = 1, then the CMPCS instruction is executed and sets Z = 0. So, the BEQ instruction is not executed and the loop terminates.

- If R2 != 0 and R3 != 0, the CMP instruction sets C = 1, then the CMPCS instruction is executed and sets Z according to whether R2 == R3. So, the BEQ instruction is executed if R2 == R3 and the loop terminates if R2 != R3.

Much faster string comparison routines are possible by loading one word of each string at a time and comparing all four bytes.

9.3.3 Long branch

A Load instruction can be used to generate a branch to anywhere in the 4GB address space. By manually setting the value of the link register (R14), a subroutine call can be made to anywhere in the address space.

```
; Long branch (and link)
    ADD    LR, PC, #4          ; set the return address to be 8 bytes
                               ; after the next instruction
    LDR    PC, [PC, #-4]       ; get the address from the next word
    DCD    function            ; store the address of the function
                               ; (DCD is an assembler directive)
return_here                    ; return to here
```

This code uses the location after the load to hold the address of the function to call. In practice, this location can be anywhere as long as it is within 4KB of the load instruction. Notice also that this code is position-independent except for the address of the function to call. Full position-independence can be achieved by storing the offset of the branch target after the load, and using an ADD instruction to add it to the PC.

9.3.4 Multi-way branches

The following code improves on the multi-way branch code shown above by using a table of addresses of functions to call:

```
; Multi-way branch
; On entry: R0 holds the branch index

    CMP     R0, #maxindex        ; checks the index is in the range
                                 ; by using an unsigned compare.
    LDRLO   PC, [PC, R0, LSL #2] ; convert the index to a word offset
                                 ; do a look up in the table put the loaded
                                 ; value into the PC and jump there
    B       IndexOutOfRange      ; jump to the error handler
    DCD     Handler0             ; DCD is an assembler directive to
    DCD     Handler1             ; store a word (in this case an
    DCD     Handler2             ; address in memory).
    DCD     Handler3
    ....
```

9.4 Load and Store Multiple instructions

Load and Store Multiple instructions are the most efficient way to manipulate blocks of data.

9.4.1 Simple block copy

This code performs a very simple block copy, 48 bytes at a time, and approaches the maximum throughput for a particular machine:

```
; Simple block copy function
; R12 points to the start of the source block
; R13 points to the start of the destination block
; R14 points to the end of the source block

loop    LDMIA    R12!, (R0-R11}          ; load 48 bytes
        STMIA    R13!, {R0-R11}          ; store 48 bytes
        CMP      R12, R14                ; reached the end yet?
        BLO      loop                    ; branch to the top of the loop
```

The source and destination must be word-aligned, and if the object to be copied is not a multiple of 48 bytes long, extra bytes are copied to bring the total to the next multiple of 48 bytes. A more sophisticated routine is needed if this extra copying is to be avoided.

9.4.2 Procedure entry and exit

This code uses Load and Store Multiple to preserve and restore the processor state during a procedure. The code assumes that registers R0 to R3 are argument registers, preserved by the caller of the function, so do not need to be preserved. R13 is also assumed to point to a full descending stack.

```
function
    STMFD    R13!, {R4 - R12, R14}       ; preserve all the local registers
                                         ; and the return address, and
                                         ; update the stack pointer.
....
Insert the function body here
....

    LDMFD    R13!, {R4 - R12, PC}        ; restore the local register, load
                                         ; the PC from the saved return
                                         ; update the stack pointer.
```

Notice that this code restores all saved registers, updates the stack pointer, and returns the caller (by loading the PC value) in a single instruction. This allows very efficient conditional return for exceptional cases from a procedure (by checking the condition with a compare instruction and then conditionally executing the Load Multiple).

9.5   Semaphore instructions

This code controls the entry and exit from a critical section of code. The semaphore instruction (SWP) does not provide a compare and conditional write facility, so this must be done explicitly. The following code achieves this by using a semaphore value to indicate that the lock is being inspected.

The code below causes the calling process to busy-wait until the lock is free. To ensure progress, three OS calls need to be made (one before each loop branch) to sleep the process if the lock cannot be accessed.

```
; Critical section entry and exit
; The code uses a process ID to identify the lock owner
; An ID of zero indicates the lock is free
; An ID of -1 indicates the lock is being inspected
; On entry:   R0 holds the address of the semaphore
;             R1 holds the ID of the process requesting the lock

          MVN    R2, #0              ; load the 'looking' value (-1) in R2
spinin    SWP    R3, R2, [R0]        ; look at the lock, and lock others out
          CMN    R3, #1              ; anyone else trying to look?
          ....
          Insert conditional OS call to sleep process here
          ....
          BEQ    spinin             ; yes, so wait our turn
          CMP    R3, #0             ; no-one looking, is the lock free?
          STRNE  R3, [R0]           ; no, then restore the previous owner
          ....
          Insert conditional OS call to sleep process here
          ....
          BNE    spinin             ; and wait again
          STR    R1, [R0]           ; otherwise grab the lock
          .....
          Insert critical code here
          .....
spinout   SWP    R3, R2, [R0]        ; look at the lock, and lock others out
          CMN    R3, #1              ; anyone else trying to look ?
          ....
          Insert conditional OS call to sleep process here
          ....
          BEQ    spinout            ; yes, so wait our turn
          CMP    R3, R1             ; check we own it
          BNE    CorruptSemaphore   ; we should have been the owner!
          MOV    R2, #0             ; load the 'free' value
          STR    R2, [R0]           ; and open the lock
```

9.6 Other code examples

The following sequences illustrate some other applications of ARM assembly language.

9.6.1 Software interrupt dispatch

This code segment dispatches *software interrupts* (SWIs) to individual handlers. For it to work, the instruction at the software interrupt vector (memory location 0x00000008) must branch to the first instruction of this code. The SWI instruction has a 24-bit field that can be used for specific SWI functions. This code also handles the 16-bit Thumb SWI instruction, which has an 8-bit SWI number field rather than a 24-bit field.

This example assumes that the code to handle each of the individual SWIs only modifies r0-r3, r12, lr and the PC. If more registers are needed, the example should be modified to include the extra registers needed in the register lists of the STMFD and LDMFD instructions. This makes the extra registers available to all of the SWI handlers, but the code will typically take longer to execute because of the extra memory accesses.

Alternatively, if only a few of the individual SWI handlers require extra registers, use extra STMFD and LDMFD instructions within those handlers. This ensures that SWIs which do not require the extra registers are not slowed down.

```
SWIHandler
    STMFD       sp!, {r0-r3,r12,lr}     ; Store the registers
    MRS         r0, spsr                ; Move SPSR into general purpose
                                        ;  register
    TST         r0, #0x20               ; Test the SPSR T bit to discover
                                        ;  ARM/Thumb state when SWI occurred
    LDRNEH      r0, [lr, #-2]           ; T bit set so load halfword (Thumb)
    BICNE       r0, r0, #0xff00         ;  and clear top 8 bits of halfword
                                        ;  (LDRH clears top 16 bits of word)
    LDREQ       r0, [lr, #-4]           ; T bit clear so load word (ARM)
    BICEQ       r0, r0, #0xff000000     ;  and clear top 8 bits of word
    CMP         r0, #MaxSWI             ; Check the SWI number is in range
    LDRLS       pc, [pc, r0, LSL #2]    ; If so, jump to the correct routine
    B           SWIOutOfRange
switable
    DCD         do_swi_0
    DCD         do_swi_1
    :
    :
do_swi_0
    .....
    Insert code to handle SWI 0 here
    .....
    LDMFD       sp!, {r0-r3,r12,pc}^    ; Restore the registers and return.
do_swi_1
    :
```

                   ARM DDI 0100E

9.6.2 Single-channel DMA transfer

The following code is an interrupt handler to perform interrupt driven input/output to memory transfers (soft DMA). The code is written as an FIQ handler, and uses the banked FIQ registers to maintain state between interrupts. Therefore this code is best situated at location `0x1C`. The entire sequence to handle a normal transfer is just four instructions. Code situated after the conditional return is used to signal that the transfer is complete.

```
LDR     r11, [r8, #IOData]    ; load port data from the I/O device
STR     r11, [r9], #4         ; store it to memory: update the pointer
CMP     r9, r10               ; reached the end?
SUBLTS  pc, lr, #4            ; no, so return
; Insert transfer complete code here
```

where:

R8   Points to the base address of the input/output device that data is read from.

IOData   Is the offset from the base address to the 32-bit data register that is read. Reading this register disables the interrupt.

R9   Points to the memory location where data is being transferred.

R10   Points to the last address to transfer to.

Of course, byte transfers can be made by replacing the load and store instructions with Load and Store byte instructions, and changing the offset in the store instruction from 4 to 1. Transfers from memory to an input/output device are made by swapping the addressing modes between the Load instruction and the Store instruction.

9.6.3 Dual-channel DMA transfer

This code is similar to the example in *Single-channel DMA transfer* on page A9-13, except that it handles two channels (which can be the input and output side of the same channel). Again, this code is written as an FIQ handler, and uses the banked FIQ registers to maintain state between interrupts. Therefore this code is best situated at location `0x1C`.

The entire sequence to handle a normal transfer is just nine instructions. Code situated after the conditional return is used to signal that the transfer is complete.

```
LDR     r13, [r8, #IOStat]       ; load status register to find ....
TST     r13, #IOPort1Active      ; .... which port caused the interrupt?
LDREQ   r13, [r8, #IOPort1]      ; load port 1 data
LDRNE   r13, [r8, #IOPort2]      ; load port 2 data
STREQ   r13, [r9], #4            ; store to buffer 1
STRNE   r13, [r10], #4           ; store to buffer 2
CMP     r9, r11                  ; reached the end?
CMPNE   r10, r12                 ; on either channel?
SUBNES  pc, lr, #4               ; return
; Insert transfer complete code here
```

where:

R8	Points to the base address of the input/output device that data is read from.
IOStat	Is the offset from the base address to a register indicating which of two ports caused the interrupt.

IOPort1Active

Is a bit mask indicating if the first port caused the interrupt (otherwise it is assumed that the second port caused the interrupt).

IOPort1,IOPort2

Are offsets to the two data registers to be read. Reading a data register disables the interrupt for that port.

R9	Points to the memory location that data from the first port is being transferred to.
R10	Points to the memory location that data from the second port is being transferred to.
R11,R12	Point to the last address to transfer to (R11 for the first port, R12 for the second).

Again, byte transfers can be made by suitably replacing the load and store instructions. Transfers from memory to an input/output device are made by swapping the addressing modes between the conditional load instructions and the conditional store instructions.

9.6.4 Interrupt prioritization

This code dispatches up to 32 interrupt sources to their appropriate handler routines. This code is intended to use the normal interrupt vector, so memory location `0x00000018` must contain an instruction that branches to the first instruction of this code.

External hardware is used to prioritize the interrupt and present the number of the highest-priority active interrupt in an input register. Interrupts are re-enabled after 10 instructions (including the branch to this code).

```
    ; first save the critical state
    ;
    SUB    r14, r14, #4              ; adjust return address before saving it
    STMFD  r13!, {r12, r14}          ; stack return address and working register
    MRS    r12, SPSR                 ; get the SPSR ...
    STMFD  r13!, {r12}               ; ... and stack that too
    ;
    ; now get the priority level of the highest priority active interrupt
    MOV    r12, #IntBase             ; get interrupt controller's base address
    LDR    r12, [r12, #IntLevel]     ; get the interrupt level (0 to 31)
    ;
    ; now read-modify-write the CPSR to enable interrupts
    MRS    r14, CPSR                 ; read the status register
    BIC    r14, r14, #0x80           ; clear the I bit (use 0x40 for the F bit)
    MSR    CPSR_c, r14               ; write it back to re-enable interrupts
    ; jump to the correct handler
    LDR    PC, [PC, r12, LSL #2]     ; and jump to the correct handler. PC base
                                     ; address points to this instruction + 8
    NOP                              ; pad so the PC indexes this table
    ;
    ; table of handler start addresses
    ;
    DCD    Priority0Handler
    DCD    Priority1Handler ........
Priority0Handler
    STMFD  r13!, {r0 - r11}          ; save working registers
    ;
    ; insert handler code here
    ;
    ........
    MRS    r12, CPSR                 ; Read-modify-write the CPSR to disable
    ORR    r12, r12, #0x80           ; interrupts (use 0x40 instead for FIQs)
    MSR    CPSR_c, r12               ; Note: Do not use r14 instead of r12. It
                                     ; will be corrupted if an interrupt occurs
    LDMFD  r13!, {r0-r12}            ; Recover the working registers and SPSR
    MSR    SPSR_cxsf, r12            ; Put the SPSR back
    LDMFD  r13!, {r12, PC}^          ; Restore last working register and return
Priority1Handler
    ........
```

where:

R13   Is assumed to point to a small Full Descending stack. The stack space required is 60 bytes
                times the maximum level to which interrupts can possibly be nested.

IntBase   Holds the base address of the interrupt handler.

IntLevel   Holds the offset (from IntBase) of the register containing the highest priority active
                interrupt.

9.6.5   Context switch

This section gives a very simple example of how to perform context switches between User mode processes,
in order to illustrate some of the instructions used for this purpose. It makes the following assumptions about
the system design:

- Context switches are performed by an IRQ handler. This handler first performs normal interrupt
  processing to identify the source of the interrupt and deal with it. The details of this are
  system-specific and are not described here. At the end of normal interrupt processing, the interrupt
  handler can choose either to return to the interrupted process, or to switch to another process.

- Only User mode context switches are to be supported. If an IRQ is allowed to occur in a privileged
  process, the IRQ handler always returns to the interrupted process.

- The normal interrupt processing code requires registers R0-R3, R12 and R14_irq to be preserved
  around it. It leaves R4-R11 unchanged, and uses R13_irq as a Full Descending stack pointer. (These
  assumptions basically mean that it can call subroutines that adhere to the standard ARM Procedure
  Calling Standard.)

- The normal interrupt processing code does not re-enable interrupts, change SPSR_irq or change to
  another processor mode, and FIQ handlers also do not re-enable interrupts. As a result, neither
  SPSR_irq nor the banked versions of R13, R14 and the SPSR belonging to the interrupted process
  are changed by execution of the normal interrupt processing code.

- Each User mode process has an associated *Process Control Block* (PCB), which stores its register
  values while it is not running. The format of a PCB is shown in Figure 9-1.

CPSR
Restart address
R0
R1
R2
R3
R4
R5
R6
R7
R8
R9
R10
R11
R12
R13
R14

Increasing addresses
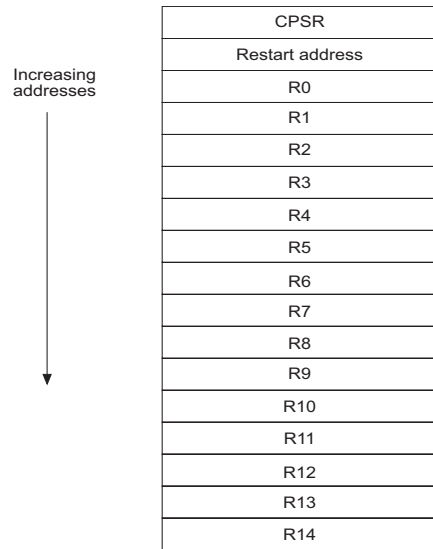
Figure 9-1 PCB layout

On entry to the IRQ handler, the following code is used to calculate the correct return address and to preserve the registers required by the normal interrupt processing code:

```
SUB     R14, R14, #4
STMFD   R13!, {R0-R3, R12, R14}
```

This is followed by the normal interrupt processing code. If this code decides to return to the interrupted process, it executes the instruction:

```
LDMFD   R13!, {R0-R3, R12, PC}^
```

This instruction is the form of LDM described in *LDM (3)* on page A4-34, and causes:

*   Registers R0-R3 and R12 to be reloaded with their values on entry to the IRQ handler, which were stored by the STMFD instruction.

*   The PC to be reloaded with the R14 value stored by the STMFD instruction, which is 4 less than the value of R14_irq on entry to the IRQ handler and so is the address of the next instruction to be executed in the interrupted process (see *Interrupt request (IRQ) exception* on page A2-19).

*   The CPSR to be reloaded from SPSR_irq, which was set to the CPSR of the interrupted process on interrupt entry and has remained unchanged since.

The values of all other registers belonging to the interrupted process were left unchanged by interrupt entry and by execution of the normal interrupt processing code, so this fully restores the context of the interrupted process.

If the normal interrupt processing code instead switches to another User mode process, it puts pointers to the PCBs of the old and new processes in R0 and R1 respectively and branches to the following code:

```
; First store the old process's User mode state to the PCB pointed to by R0.
MRS     R12, SPSR               ; Get CPSR of interrupted process
STR     R12, [R0], #8           ; Store CPSR to PCB, point R0 at
                                ;   PCB location for R0 value
LDMFD   R13!, {R2, R3}          ; Reload R0/R1 of interrupted
                                ;   process from stack
STMIA   R0!, {R2, R3}           ; Store R0/R1 values to PCB, point
                                ;   R0 at PCB location for R2 value
LDMFD   R13!, {R2, R3, R12, R14} ; Reload remaining stacked values
STR     R14, [R0, #-12]         ; Store R14_irq, the interrupted
                                ;   process's restart address
STMIA   R0, {R2-R14}^           ; Store user R2-R14 - see Note 1

; Then load the new process's User mode state and return to it.
LDMIA   R1!, {R12, R14}         ; Put interrupted process's CPSR
MSR     SPSR_fsxc, R12          ;   and restart address in SPSR_irq
                                ;   and R14_irq
LDMIA   R1, {R0-R14}^           ; Load user R0-R14 - see Note 2
NOP                             ; Note: Cannot use banked register
                                ;   immediately after User mode LDM
MOVS    PC, R14                 ; Return to address in R14_irq,
                                ;   with SPSR_irq -> CPSR transfer
```

────── **Note** ──────

1.   This instruction is an example of the form of STM described in *STM (2)* on page A4-86. It stores the registers R2, R3, ..., R12, R13_usr, R14_usr to the correct places in the PCB.

2.   This instruction is an example of the form of LDM described in *LDM (2)* on page A4-32. It loads the registers R0, R1, ..., R12, R13_usr, R14_usr from the correct places in the PCB.

Chapter A10
Enhanced DSP Extension

This chapter describes the enhanced DSP additions to the ARM programmer's model and instruction set, included in E variants of ARM architecture versions 5 and above. It contains the following sections:

- *About the enhanced DSP instructions* on page A10-2
- *Saturated integer arithmetic* on page A10-3
- *Saturated Q15 and Q31 arithmetic* on page A10-4
- *The Q flag* on page A10-5
- *Enhanced DSP instructions* on page A10-6
- *Alphabetical list of enhanced DSP instructions* on page A10-8.

10.1 About the enhanced DSP instructions

Many *digital signal processing* (DSP) algorithms operate on arrays of 16-bit data, where the 16-bit value is to be interpreted as a signed fixed-point number with 15 binary places. Such values are sometimes called *Q15 numbers*, and represent numeric values ranging from –1 up to $+1 - 2^{-15}$.

To preserve accuracy, intermediate values in these algorithms are often calculated as *Q31 numbers*, which are similar but have 32 bits and 31 binary places. Also, in order to avoid *spikes* in the output from the algorithm if numeric overflow occurs, arithmetic on Q15 and Q31 values is normally *saturated*. This means that if overflow occurs, the result is set to the most positive or most negative possible value depending on the direction of overflow. In contrast, normal integer arithmetic will wrap around modulo 2^{32}.

Performing saturated arithmetic on Q15 and Q31 numbers is possible using the standard ARM instruction set, but takes roughly 5-10 instructions per arithmetic operation. The enhanced DSP instructions described in this chapter include instructions to perform such arithmetic considerably more quickly, using 1-2 instructions per arithmetic operation.

In order to maximize the performance of DSP algorithms, it is also important that data and coefficient values should be loaded and stored efficiently. The enhanced DSP instructions therefore also include instructions to assist with this loading and storing.

Finally, the enhanced DSP instructions include coprocessor instructions which transfer 64 bits of data directly between the ARM processor and the coprocessor, in order to assist the design of coprocessors which will further improve the performance of DSP algorithms.

10.2   Saturated integer arithmetic

When viewed as a signed number, the value of a general-purpose register lies in the range from -2^{31} (or `0x80000000`) to $+2^{31} - 1$ (or `0x7FFFFFFF`). If an addition or subtraction is performed on such numbers and the correct mathematical result lies outside this range, it would require more than 32 bits to represent. In these circumstances, the surplus bits are normally discarded, which has the effect that the result obtained is equal to the correct mathematical result reduced modulo 2^{32}.

For example, `0x60000000` could be used to represent $+3 \times 2^{29}$ as a signed integer. If you add this number to itself, you get $+3 \times 2^{30}$, which lies outside the representable range, but could be represented as the 33-bit signed number `0x0C0000000`. The actual result obtained will be the rightmost 32 bits of this, which are `0xC0000000`. This represents -2^{30}, which is smaller than the correct mathematical result by 2^{32}, and does not even have the same sign as the correct result.

This kind of inaccuracy is unacceptable in many DSP applications. For example, if it occurred while processing an audio signal, the abrupt change of sign would be likely to result in a loud click. To avoid this sort of effect, many DSP algorithms use *saturated* signed arithmetic. This modifies the way normal integer arithmetic behaves as follows:

* If the correct mathematical result lies within the available range from -2^{31} to $+2^{31} - 1$, the result of the operation is equal to the correct mathematical result.

* If the correct mathematical result is greater than $+2^{31} - 1$ and so overflows the upper end of the representable range, the result of the operation is equal to $+2^{31} - 1$.

* If the correct mathematical result is less than -2^{31} and so overflows the lower end of the representable range, the result of the operation is equal to -2^{31}.

Put another way, the result of a saturated arithmetic operation is the closest representable number to the correct mathematical result of the operation.

The enhanced DSP instructions support saturated signed 32-bit integer additions and subtractions, by use of the `QADD` and `QSUB` instructions. Variants of these instructions (`QDADD` and `QDSUB`) perform a saturated doubling of one of the operands before the saturated addition or subtraction.

Saturated integer multiplications are not supported, because the product of two values of widths A and B bits never overflows an (A+B)-bit destination.

10.3　Saturated Q15 and Q31 arithmetic

A 32-bit signed value can be treated as having a binary point immediately after its sign bit. This is equivalent to dividing its signed integer value by 2^{31}, so that it can now represent numbers from -1 to $+1 - 2^{-31}$. When a 32-bit value is used to represent a fractional number in this fashion, it is known as a *Q31 number*.

Saturated additions, subtractions, and doublings can be performed on Q31 numbers using the same instructions as are used for saturated integer arithmetic, since everything is simply scaled down by a factor of 2^{-31}.

Similarly, a 16-bit value can be treated as having a binary point immediately after its sign bit, which effectively divides its signed integer value by 2^{15}. When a 16-bit value is used in this fashion, it can represent numbers from -1 to $+1 - 2^{-15}$ and is known as a *Q15 number*.

If two Q15 numbers are multiplied together as integers, the resulting integer needs to be scaled down by a factor of $2^{-15} \times 2^{-15} == 2^{-30}$. For example, multiplying the Q15 number `0x8000` (representing -1) by itself using an integer multiplication instruction yields the value `0x40000000`, which is 2^{30} times the desired result of $+1$.

This means that the result of the integer multiplication instruction is not quite in Q31 form. To get it into Q31 form, it must be doubled, so that the required scaling factor becomes 2^{-31}. Furthermore, it is possible that the doubling will cause integer overflow, so the result should in fact be doubled with saturation. In particular, the result `0x40000000` from the multiplication of `0x8000` by itself should be doubled with saturation to produce `0x7FFFFFFF` (the closest possible Q31 number to the correct mathematical result of $-1 \times -1 == +1$). If it were doubled without saturation, it would instead produce `0x80000000`, which is the Q31 representation of -1.

To implement a saturated Q15 $\times$ Q15 $\rightarrow$ Q31 multiplication, therefore, an integer multiply instruction should be followed by a saturated integer doubling. The latter can be performed by a `QADD` instruction adding the multiply result to itself.

Similarly, a saturated Q15 $\times$ Q15 $+$ Q31 $\rightarrow$ Q31 multiply-accumulate can be performed using an integer multiply instruction followed by the use of a `QDADD` instruction.

Some other examples of arithmetic on Q15 and Q31 numbers are described in the Usage sections for the individual instructions.

10.4 The Q flag

The enhanced DSP extension incorporates a mechanism to determine whether saturation or overflow has occurred in the course of a calculation. Bit[27] of the CPSR is a *sticky overflow* flag, also known as the Q flag. This flag is set to 1 if any of the following occurs:

- Saturation of the addition result in a QADD or QDADD instruction
- Saturation of the subtraction result in a QSUB or QDSUB instruction
- Saturation of the doubling intermediate result in a QDADD or QDSUB instruction
- Signed overflow during an SMLA<x><y> or SMLAW<y> instruction.

——— **Note** ———

The Q flag is not affected by overflow during any other arithmetic instruction, such as ADD, SUB, or MLA.

The Q flag is *sticky* in that once it has been set to 1, it is not affected by whether subsequent calculations saturate and/or overflow. Its intended usage is:

1. Use an MSR CPSR_f,#0 instruction to clear the Q flag (this also clears the condition code flags).

2. Perform a sequence of calculations.

3. Use an MRS Rn,CPSR instruction to read the CPSR, then test the value of the Q flag. If it is still 0, none of the above types of saturation or overflow occurred during step 2. Otherwise, at least one instance of saturation or overflow occurred.

Each SPSR also has a Q flag into which the CPSR Q flag is copied when an exception occurs as part of the general CPSR → SPSR copy performed on exception entry. Similarly, the exception return instructions which atomically return to the correct address and perform an SPSR → CPSR transfer will copy the SPSR Q flag back to the CPSR. Between them, these ensure that the value of the Q flag is not changed by an interrupt or other exception. For more details of this, see *Exceptions* on page A2-13.

Except as described above, the only instructions that affect or are affected by the Q flags are MSR instructions which write to the flags byte of the destination PSR, and MRS instructions.

10.5 Enhanced DSP instructions

The enhanced DSP instructions can be divided into the following groups:

- *Integer multiply and multiply-accumulate instructions* on page A10-6
- *Saturated addition and subtraction instructions* on page A10-7
- *Two-word load and store instructions* on page A10-7
- *Cache preload instruction* on page A10-7
- *Two-word coprocessor register transfer instructions* on page A10-7.

10.5.1 Integer multiply and multiply-accumulate instructions

The enhanced DSP instructions augment the normal integer multiply and multiply-accumulate instructions with versions designed to work on 16-bit signed data. These form one or both of the operands to the multiplication by taking either the bottom half (bits[15:0]) or the top half (bits[31:16]) of a source register and sign-extending it, while ignoring the other half of the source register.

These instructions allow multiplications to be performed on 16-bit data values that are held packed in registers, without the need to unpack them. The data values can be loaded in this format by `LDR` or `LDM` instructions, which generally results in more efficient use of memory bandwidth than would be obtained by the use of `LDRH` or `LDRSH` instructions.

The instructions available are:

- $16 \times 16 \rightarrow 32$ bit signed multiply instructions `SMUL<x><y>`. These instructions are very similar to the normal `MUL` instruction, apart from the fact that their operands are generated as described above.

- $16 \times 16 + 32 \rightarrow 32$ bit signed multiply-accumulate instructions `SMLA<x><y>`. These instructions are very similar to the normal `MLA` instruction, apart from the fact that their operands are generated as described above.

- $32 \times 16 \rightarrow 32$ bit signed multiply instructions `SMULW<y>`. These instructions produce the upper 32 bits of the 48-bit product.

- $32 \times 16 + 32 \rightarrow 32$ bit signed multiply-accumulate instructions `SMLAW<y>`. The multiplication in these instructions produces the upper 32 bits of the 48-bit product, and these 32 bits are then added to the accumulate value.

- $16 \times 16 + 64 \rightarrow 64$ bit signed multiply instructions `SMLAL<x><y>`. These instructions are very similar to the normal `SMLAL` instruction, apart from the fact that their operands are generated as described above.

In all cases, `<y>` is either the letter B or T, depending on whether the bottom or top half of the second operand register should be used. Where the first operand is a sign-extended 16-bit value, `<x>` specifies whether the bottom or top half of the first operand register should be used in a similar fashion.

10.5.2 Saturated addition and subtraction instructions

Instructions are provided to perform saturated integer addition and subtraction, with an optional saturated doubling of one operand. These instructions can be used to perform saturated integer, Q15, and Q31 arithmetic, as described in *Saturated integer arithmetic* on page A10-3 and *Saturated Q15 and Q31 arithmetic* on page A10-4.

The instructions are:

QADD   Performs a saturated integer addition.

QDADD   Performs a saturated integer doubling of one operand followed by a saturated integer addition with the other operand.

QSUB   Performs a saturated integer subtraction.

QDSUB   Performs a saturated integer doubling of one operand followed by a saturated integer subtraction from the other operand.

10.5.3 Two-word load and store instructions

Load and store instructions are provided which can be used to transfer two adjacent words of memory to or from any of the register pairs {R0,R1}, {R2,R3}, {R4,R5}, {R6,R7}, {R8,R9}, {R10,R11}, or {R12,R13}. These instructions use the same addressing modes as LDRH, LDRSB, LDRSH, and STRH, and thus allow a larger addressing range than is possible with the LDM and STM instructions. A restriction is that the doubleword formed by the two adjacent words of memory is required to be doubleword-aligned.

The instructions are:

LDRD   Loads two words.

STRD   Stores two words.

10.5.4 Cache preload instruction

The PLD instruction can be used to tell the memory system that an access to the data at a specified memory address is likely to occur soon. A memory system that recognizes this request will bring the data into cache, so that when the access actually occurs, any delay caused by a cache miss is reduced or eliminated.

10.5.5 Two-word coprocessor register transfer instructions

Two additional generic coprocessor instructions are defined. These are similar to the normal MCR and MRC generic coprocessor instructions, but transfer two ARM register values to or from the coprocessor rather than just one. They can be used when designing coprocessors which require more ARM-coprocessor bandwidth than is available from the MCR and MRC instructions.

The instructions are:

MCRR   Transfers two ARM register values to a coprocessor.

MRRC   Transfers values from a coprocessor to two ARM registers.

10.6 Alphabetical list of enhanced DSP instructions

This section describes each enhanced DSP instruction in alphabetical order, using the same format and conventions as are used in the descriptions of normal ARM instructions (see *Alphabetical list of ARM instructions* on page A4-2).

10.6.1 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	
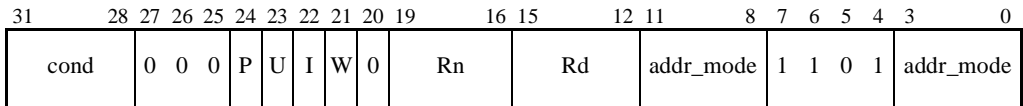
The LDRD instruction loads a pair of ARM registers from two consecutive words of memory. The pair of registers is restricted to being an even-numbered register and the odd-numbered register that immediately follows it (for example, R10 and R11).

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 A3-5. 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 A5-34. 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 writeback*).

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

E variants of version 5 and above, excluding 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 writeback 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 A2-17.

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 B2 *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.

10.6.2   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
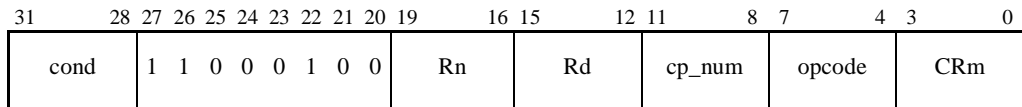
The MCRR instruction passes the values of ARM registers <Rd> and <Rn> to the coprocessor whose number is cp_num. If no coprocessors indicate that they can execute the instruction, an undefined instruction exception is generated.

Syntax

```
MCRR{<cond>}  <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 A3-5. If <cond> is omitted, the AL (always) condition is used.

<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

E variants of version 5 and above, excluding ARMv5TExP

Exceptions

Undefined instruction

Operation

```
if ConditionPassed(cond) then
    send Rd value to Coprocessor[cp_num]
    send Rn value to Coprocessor[cp_num]
```

Usage

MCRR is used to initiate coprocessor operations that depend on values in 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 may 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 trap.

Order of transfers

If a coprocessor uses these instructions, it will define 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.

10.6.3 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	
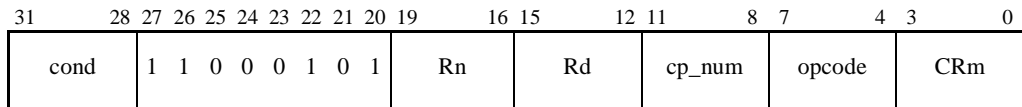
The MRRC instruction causes the coprocessor whose number is cp_num to transfer values to two ARM registers <Rd> and <Rn>. If no coprocessors indicate that they can execute the instruction, an undefined instruction exception is generated.

Syntax

```
MRRC{<cond>}  <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 A3-5. If <cond> is omitted, the AL (always) condition is used.

<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

E variants of version 5 and above, excluding ARMv5TExP

Exceptions

Undefined instruction

Operation

```
if ConditionPassed(cond) then
    Rd = first value from Coprocessor[cp_num]
    Rn = second value from Coprocessor[cp_num]
```

Usage

MRRC is used to initiate coprocessor operations that write 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 may 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 trap.

Order of transfers

If a coprocessor uses these instructions, it will define 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.

10.6.4 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
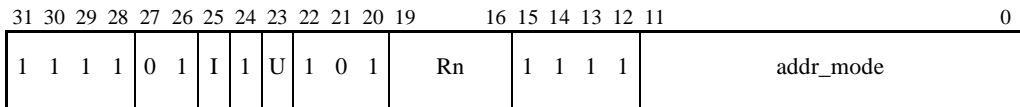
The `PLD` instruction signals the memory system that memory accesses from a specified address are likely in the near future. The memory system can respond by taking actions which are expected to speed up the memory accesses when they do occur, such as pre-loading the cache line containing the specified 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 A5-18. 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

E variants of version 5 and above, excluding 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, this instruction cannot be executed conditionally.

Writeback   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 B2 *The System Control Coprocessor*), it will not generate an alignment exception for any PLD instruction.

10.6.5 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	
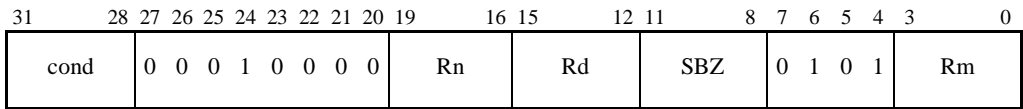
The QADD instruction performs integer addition, saturating the result to the 32-bit signed integer range $-2^{31} \leq x \leq 2^{31} - 1$. If saturation actually occurs, the instruction sets the Q flag in the CPSR.

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 A3-5. If <cond> is omitted, the AL (always) condition is used.

<Rd>   Specifies the destination register of the instruction.

<Rm>   Specifies the register that contains the first operand for the saturated addition.

<Rn>   Specifies the register that contains the second operand for the saturated addition.

Architecture version

E variants of version 5 and above

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, this instruction can be used 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   The QADD instruction does not affect the N, Z, C, or V flags.

10.6.6   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	
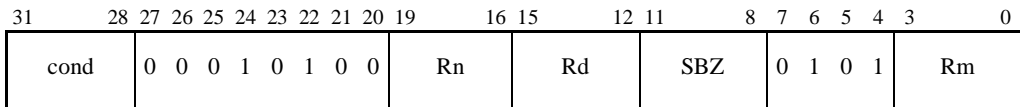
The QDADD instruction doubles its second operand, then adds the result to its first operand. Both the doubling and the addition have their results saturated to the 32-bit signed integer range $-2^{31} \leq x \leq 2^{31} - 1$. If saturation actually occurs in either operation, the instruction sets the 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 A3-5. If <cond> is omitted, the AL (always) condition is used.
<Rd>	Specifies the destination register of the instruction.
<Rm>	Specifies the register that contains the first operand for the saturated addition.
<Rn>	Specifies the register whose value is to be doubled, saturated, and used as the second operand for the saturated addition.

Architecture version

E variants of version 5 and above

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.

10.6.7   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	

The QDSUB instruction 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 actually occurs in either operation, the instruction 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 A3-5. If <cond> is omitted, the AL (always) condition is used.
<Rd>	Specifies the destination register of the instruction.
<Rm>	Specifies the register that contains the first operand for the saturated subtraction.
<Rn>	Specifies the register whose value is to be doubled, saturated, and used as the second operand for the saturated subtraction.

Architecture version

E variants of version 5 and above

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.

10.6.8 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

The QSUB instruction performs integer subtraction, saturating the result to the 32-bit signed integer range $-2^{31} \leq x \leq 2^{31} - 1$. If saturation actually occurs, the instruction 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 A3-5. If <cond> is omitted, the AL (always) condition is used.

<Rd>   Specifies the destination register of the instruction.

<Rm>   Specifies the register that contains the first operand for the saturated subtraction.

<Rn>   Specifies the register that contains the second operand for the saturated subtraction.

Architecture version

E variants of version 5 and above

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   The QSUB instruction does not affect the N, Z, C, or V flags.

10.6.9 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	
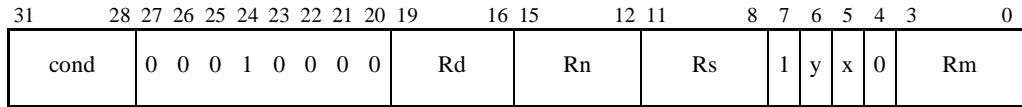
The SMLA<x><y> instructions (SMLABB, SMLABT, SMLATB and SMLATT) perform a signed multiply-accumulate operation. The multiply acts on two signed 16-bit quantities, taken from either the bottom or the top half of their respective source registers. The other halves of these source registers are ignored. 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 A3-5. If <cond> is omitted, the AL (always) condition is used.

<Rd>   Specifies the destination register of the instruction.

<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

E variants of version 5 and above

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 will set 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.

10.6.10 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	
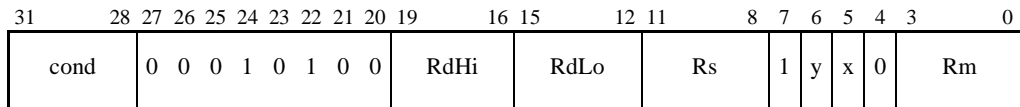
The SMLAL<x><y> instructions (SMLALBB, SMLALBT, SMLALTB and SMLALTT) perform a signed multiply-accumulate operation. The multiply acts on two signed 16-bit quantities, taken from either the bottom or the top half of their respective source registers. The other halves of these source registers are ignored. 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 A3-5. 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

E variants of version 5 and above

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> will hold 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.

10.6.11 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	
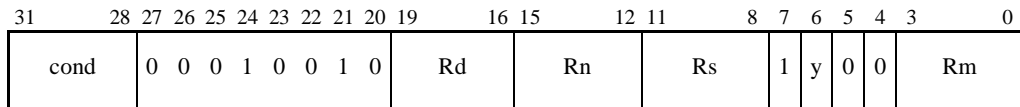
The SMLAW<y> instructions (SMLAWB and SMLAWT) perform a signed multiply-accumulate operation. The multiply acts on a signed 32-bit quantity and a signed 16-bit quantity, with the latter being taken from either the bottom or the top half of its source register. The other half of the second source register 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 A3-5. If <cond> is omitted, the AL (always) condition is used.
<Rd>	Specifies the destination register of the instruction.
<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

E variants of version 5 and above

Exceptions

None

 ARM DDI 0100E

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 its 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 A10-24 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.

10.6.12 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	
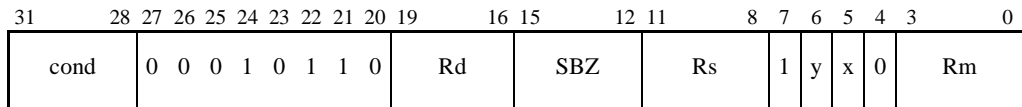
The SMUL<x><y> instructions (SMULBB, SMULBT, SMULTB, and SMULTT) perform a signed multiply operation. The multiply acts on two signed 16-bit quantities, taken from either the bottom or the top half of their respective source registers. The other halves of these source registers are ignored. No overflow 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 A3-5. If <cond> is omitted, the AL (always) condition is used.

<Rd>   Specifies the destination register of the instruction.

<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

E variants of version 5 and above

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 the QADD, QDADD, and QDSUB instructions to perform multiplies, multiply-accumulates, and multiply-subtracts on Q15 numbers. See the *Usage* sections on page A10-17, page A10-19, and page A10-21 for examples.

Notes

Use of R15   Specifying R15 for register <Rd>, <Rm>, or <Rs> has UNPREDICTABLE results.

Condition flags   The SMUL<x><y> instructions do not affect the N, Z, C, V, or Q flags.

10.6.13 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	
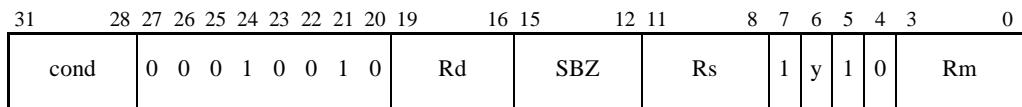
The SMULW<y> instructions (SMULWB and SMULWT) perform a signed multiply operation. The multiply acts on a signed 32-bit quantity and a signed 16-bit quantity, with the latter being taken from either the bottom or the top half of its source register. The other half of the second source register is ignored. 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 A3-5. If <cond> is omitted, the AL (always) condition is used.

<Rd>   Specifies the destination register of the instruction.

<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

E variants of version 5 and above

Exceptions

None

 ARM DDI 0100E

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 the QADD, QDADD, and QDSUB instructions to perform multiplies, multiply-accumulates and multiply-subtracts between Q31 and Q15 numbers. See the *Usage* sections on page A10-17, page A10-19, and page A10-21 for examples.

Notes

Use of R15 Specifying R15 for register <Rd>, <Rm>, or <Rs> has UNPREDICTABLE results.

Condition flags The SMULW<y> instructions do not affect the N, Z, C, V, or Q flags.

10.6.14 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
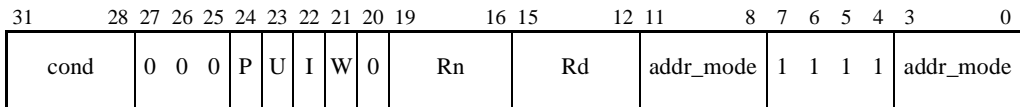
The STRD instruction stores a pair of ARM registers to two consecutive words of memory. The pair of registers is restricted to being an even-numbered register and the odd-numbered register that immediately follows it (for example, R10 and R11).

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 A3-5. If <cond> is omitted, the AL (always) condition is used.

<Rd>   Specifies the even-numbered register which will be stored to the memory word addressed by <addressing_mode>. The immediately following odd-numbered register will be 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 A5-34. 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 writeback*).

                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

E variants of version 5 and above, excluding 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 writeback 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 A2-17.

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 B2 *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.

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
- *System-level issues* on page B1-4.

1.1 About the memory system

ARM processors are used in a wide range of embedded systems and other applications. 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.

The range of memory-mapped I/O devices that can be used adds further to the variety of systems based on ARM processors.

Most systems need to initialize and control their memory system facilities in various ways, such as:

- Enabling a cache, to ensure that its performance benefits are realized.

- Setting up the virtual-to-physical address mapping for a virtual memory system.

- Restricting access to memory regions.

- Ensuring that the correct accesses to memory-mapped I/O devices occur, and that they occur at the correct times. (This usually happens automatically in the simplest memory systems, but caches and other facilities in more complex systems can interfere with it.)

The standard way to perform memory system control in ARM-based systems is to use coprocessor 15 (CP15), which is also known as the *System Control coprocessor*. Chapter B2 *The System Control Coprocessor* provides a top-level overview of this coprocessor.

The rest of *Part B* (Chapter B3 through Chapter B6) describes standardized ways to perform a variety of memory system control operations using CP15:

Chapter B3 *Memory Management Unit*

> Describes 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).

Chapter B4 *Protection Unit*

> Describes a simpler *Protection Unit* system which is suitable for many applications which do not require the full facilities provided by the MMU memory system.

Chapter B5 *Caches and Write Buffers*

> Describes facilities to control caches and write buffers. These are common to the MMU and Protection Unit systems

Chapter B6 *Fast Context Switch Extension*

> Describes the Fast Context Switch Extension, which facilitates fast switching between up to 128 processes executing in separate process blocks, each of size up to 32MB.

—— **Note** ——

Because of the wide variety of systems based on ARM processors, all functionality described in *Part B* might be inappropriate to any given system. Furthermore, some ARM processors have implemented functions in a different manner to the one described here. Because of this, the datasheet or Technical Reference Manual for a particular ARM processor is the definitive source for its memory and system control facilities.

Part B therefore does not attempt to specify absolute requirements on the functionality of the System Control coprocessor or other memory system components. Instead, it contains guidelines which, if followed:

• mean that the system is more likely to be compatible with existing and future ARM software.

• probably make it easier to port incompatible software to the system.

In order to provide an adequate description of the range of memory and system facilities on existing ARM implementations, *Part B* describes a number of options that will not be used on new ARM implementations. For information on the rules that must be followed by new implementations of the memory and system architectures, contact ARM Ltd.

The fact that *Part B* describes a broad range of facilities, many of which are used only on some existing ARM implementations, also means that architecture version numbers for the memory and system architectures would not be helpful or descriptive. They are therefore not used.

1.2 System-level issues

This section lists a number of general and operating-system issues that the system designer needs to address when using an ARM processor.

1.2.1 Memory systems, write buffers and caches

ARM processors and software are designed to be connected to a byte-addressed memory. Word and halfword accesses to the memory ignore the alignment of the address and access the naturally-aligned value that is addressed (so a memory access ignores address bits 0 and 1 for word access, and ignores bit 0 for halfword accesses). The endianness of the ARM processor should normally match that of the memory system, or be configured to match it before any non-word accesses occur (when the endianness is configurable and CP15 is implemented, bit[7] of CP15 register 1 controls the endianness).

Memory that is used to hold programs and data should be marked as follows:

- Main (RAM) memory is normally set as cachable and bufferable.

- ROM memory is normally set as cachable, and should be marked as read only, so the bufferable attribute is not used and should be 1.

Write buffers

Some ARM implementations incorporate a merging write buffer that subsumes multiple writes to the same location into a single write to main memory. Furthermore, some write buffers re-order writes, so that writes are issued to memory in a different order to the order in which they are issued by the processor. Therefore, I/O locations should 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, memory aborts can only be signaled 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 a parity error from main memory) must be handled by other methods (typically by raising an interrupt).

Caches

Frame buffers can be cachable, but frame buffers on writeback cache implementations must be copied back to memory after the frame buffer has been updated. Frame buffers can be bufferable, but again 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 not supported. If memory data is to be shared between multiple bus masters without taking special software measures to ensure coherency, then the data must be mapped as:

- uncachable to ensure that all reads access main memory

- unbufferable to ensure that all write access main memory.

Alternatively, using software, you can manage the coherence of data buffers that are read or written by another bus master by:

- cleaning data from writeback 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 relevant data from caches when the buffer is being read after the other bus master has written the buffer.

You can use an uncached, unbuffered semaphore to maintain synchronization between multiple bus masters (see *Semaphores* on page B1-6).

For implementations with writeback caches, all dirty cache data must be written back before any alterations are made to the MMU page tables, to ensure that cache line write back can use the page tables to form the correct physical address for the transfer.

You can index caches using either virtual or physical addresses. Physical pages must only be mapped into a single virtual page, otherwise the result is UNPREDICTABLE. ARM processors do not normally provide coherence between multiple virtual copies of a single physical page.

Some ARM implementations support separate instruction and data caches. Coherence between the data and instruction caches is not necessarily maintained in hardware, so if the instruction stream is written, the instruction cache and data cache must be made coherent. This can entail:

- cleaning the data cache (storing dirty data to memory)
- draining the write buffer (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.

1.2.2 Interrupts

ARM processors implement fast and normal levels of interrupt. Both interrupts are signaled 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.

For more information, see *Exceptions* on page A2-13.

Canceling interrupts

It is the responsibility of software (the interrupt handler) to ensure that the cause of an interrupt is canceled (no longer signaled to the processor) before interrupts are re-enabled (by clearing the I and/or F bit in the CPSR). Interrupts can be canceled with any instruction that might make an external data bus access, meaning any load or store, a swap, or any coprocessor instruction.

Canceling an interrupt via an instruction fetch is UNPREDICTABLE.

Canceling 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 canceled before interrupts are re-enabled. This allows a device connected to a remote I/O bus to operate correctly.

1.2.3 Semaphores

The Swap and Swap Byte instructions have predictable behavior when used in 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.

Semaphores placed in uncachable/bufferable memory regions have UNPREDICTABLE results. Semaphores placed in cachable/unbufferable memory regions have UNPREDICTABLE results.

Chapter B2
The System Control Coprocessor

This chapter describes coprocessor 15, the System Control coprocessor. It contains the following sections:

2.1 About the System Control coprocessor

All of the standard memory and system facilities are controlled by coprocessor 15 (CP15), which is therefore called 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 B3 *Memory Management Unit* is also controlled by page tables in memory.

If none of the standard memory and system facilities are implemented in a system, the System Control coprocessor might not be present. In this case, no coprocessor accepts CP15 instructions, and so all such instructions are UNDEFINED.

However, new implementations of the memory and system architectures must implement the System Control coprocessor, and must follow some additional rules about which facilities are implemented. For details of these rules, contact ARM Ltd.

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.

2.2 Registers

The System Control coprocessor can contain up to 16 primary registers, each of which is 32 bits long. For some of these, additional bits in the register access instructions are used to identify a specific version of the register and/or specific types of access to the register, so the number of physical 32-bit registers in CP15 can be more than 16. However, 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.

2.2.1 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.

All CP15 `CDP`, `LDC` and `STC` instructions are UNDEFINED.

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	SBZ	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>}
MRC{<cond>}  p15, 0, <Rd>, <CRn>, <CRm>{, <opcode2>}
```

where:

`<cond>`	This is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-5. 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 always 0b000 in valid CP15 instructions. If they are not 0b000, the instruction is 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.

These `MCR` and `MRC` instructions can only be used while the processor is in a privileged mode. If they are executed while the processor is in User mode, an Undefined Instruction exception occurs.

——— **Note** ———

If access to some 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 A2-28 are examples of such SWIs.

2.2.2 Primary register allocation

Table 2-1 shows the allocation of the primary registers of the System Control coprocessor.

Table 2-1 Primary register allocation

Reg	Generic use	Specific uses	Details in
0	ID codes (read-only)	ID and Cache type	*Register 0: ID codes* on page B2-6
1	Control bits (read/write)	Miscellaneous control bits	*Register 1: Control register* on page B2-13
2	Memory protection and control	MMU: Translation table base PU: Cachability bits	*Register 2: Translation table base* on page B3-23 *Register 2: Cachability bits* on page B4-6
3	Memory protection and control	MMU: Domain access control PU: Bufferability bits	*Register 3: Domain access control* on page B3-24 *Register 3: Bufferability bits* on page B4-6
4	Memory protection and control	MMU: Reserved PU: Reserved	*Register 4: Reserved* on page B3-24 *Registers 4, 8, 10: Reserved* on page B4-7
5	Memory protection and control	MMU: Fault status PU: Access permission bits	*Register 5: Fault status* on page B3-24 *Register 5: Access permission bits* on page B4-7
6	Memory protection and control	MMU: Fault address PU: Protection area control	*Register 6: Fault address* on page B3-25 *Register 6: Protection area control* on page B4-8
7	Cache and write buffer	Cache/write buffer control	*Register 7: Cache functions* on page B5-15
8	Memory protection and control	MMU: TLB control PU: Reserved	*Register 8: TLB functions* on page B3-25 *Registers 4, 8, 10: Reserved* on page B4-7
9	Cache and write buffer	Cache lockdown	*Register 9: Cache lockdown* on page B5-18
10	Memory protection and control	MMU: TLB lockdown PU: Reserved	*Register 10: TLB lockdown* on page B3-27 *Registers 4, 8, 10: Reserved* on page B4-73
11	Reserved	-	-
12	Reserved	-	-
13	Process ID	Process ID	*Register 13: Process ID* on page B6-6
14	Reserved	-	-
15	IMPLEMENTATION DEFINED	IMPLEMENTATION DEFINED	Implementation documents

2.3 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 2-2, 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 2-2 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 B2-9
other	Reserved (see main text)	-

It is recommended that all the ID registers in Table 2-2 are implemented, but only the main ID register (<opcode2> == 0) is mandatory. Whether or not other ID registers are implemented is IMPLEMENTATION DEFINED.

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.

2.3.1 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.

For historical reasons, there are three distinct ways in which the CP15 register 0 ID code might need to be interpreted. To determine which to use, look at bits[15:12] of the ID code:

- if they are 0x0, this indicates a pre-ARM7 processor
- if they are 0x7, this indicates that the processor is in the ARM7 family
- otherwise, a more recent processor family than ARM7 is involved.

Post-ARM7 processors

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 (all other values of the architecture code are reserved by ARM Ltd):

    `0x1`   Architecture 4

    `0x2`   Architecture 4T

    `0x3`   Architecture 5

    `0x4`   Architecture 5T

    `0x5`   Architecture 5TE.

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. The following codes are defined (all other values of the architecture code are reserved by ARM Ltd):

    `0x41`   = A (ARM Ltd)

    `0x44`   = D (Digital Equipment Corporation).

    `0x69`   = i (Intel Corporation).

ARM7 family processors

If bits[15:12] of the ID code are 0x7, the ID code is interpreted as follows:
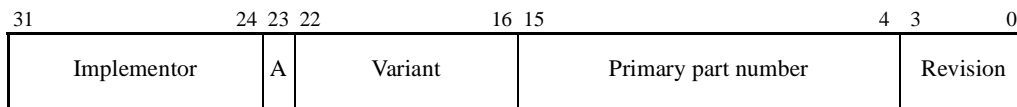
31 24	23 22	16 15	4 3 0
Implementor	A	Variant	Primary part number	Revision

Bits[3:0]   Contain the IMPLEMENTATION DEFINED revision number for the processor.

Bits[15:4]   Contain an 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 processor is involved:

0   Architecture 3

1   Architecture 4T.

Bits[31:24]   Contain an implementor code. See *Post-ARM7 processors* for these codes.

Pre-ARM7 processors

Four processors prior to ARM7 use ID codes in which bits[15:12] are 0x0, and no further processors will be allocated such ID codes. They are interpreted as a 28-bit processor ID and a 4-bit revision number:

31	4 3 0
Processor ID	Revision

The processor ID values are as follows:

0x4156030   ARM3 (Architecture 2)

0x4156060   ARM600 (Architecture 3)

0x4156061   ARM610 (Architecture 3)

0x4156062   ARM620 (Architecture 3).

2.3.2 Cache Type register

If present, 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
- how it can be cleaned efficiently (in the case of a write-back cache)
- whether cache lock-down is supported.

See *Types of cache* on page B5-5 for a discussion of these details.

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. See Table 2-3 on page B2-9 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 B2-10 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 B2-10 for details of the encoding.

Table 2-3 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
0b0010	Write-back	Register 7 operations	Not supported
0b0110	Write-back	Register 7 operations	Supported, format A
0b0111	Write-back	Register 7 operations	Supported, format B

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 functions on page B5-15 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 2-3, see *Register 9: Cache lockdown* on page B5-18.

2.3.3 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	0		size			assoc		M		len
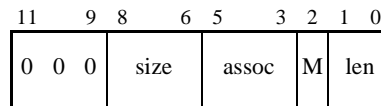
Bits[11:9] are reserved for future expansion.

The size of the cache is determined by the size field and M bit, as shown in Table 2-4.

Table 2-4 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
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 2-5.

Table 2-5 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 M bit, as shown in Table 2-6.

Table 2-6 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 B5-4, 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.

 ARM DDI 0100E

2.4   Register 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 may be useful in some circumstances is that 0 can be written to this register to restore it to its reset state.

31 16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
UNP/SBZP	L4	RR	V	I	Z	F	R	S	B	L	D	P	W	C	A	M
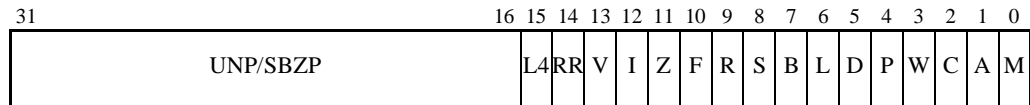
Reading from CP15 register 1 reads the control bits. The `<CRm>` and `<opcode2>` fields are ignored when reading CP15 register 1, and should be zero.

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 or Protection Unit, this bit reads as 0 and ignores writes.

A (bit[1])   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 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 and ignores writes. If the cache cannot be disabled, this bit reads as 1 and ignores writes.

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 reads as 0 and ignores writes. If the write buffer cannot be disabled, this bit reads as 1 and ignores writes.

P (bit[4])   On ARM processors which support 26-bit backwards-compatibility configurations, this bit controls the PROG32 configuration signal:

0 = Exception handlers entered in 26-bit modes

1 = Exception handlers entered in 32-bit modes.

For details, see *32-bit and 26-bit configuration* on page A8-10.

On ARM processors which do not support 26-bit backwards-compatibility configurations, this bit reads as 1 and ignores writes.

D (bit[5])   On ARM processors which support 26-bit backwards-compatibility configurations, this bit controls the DATA32 configuration signal:

0 = 26-bit address exception checking enabled

1 = 26-bit address exception checking disabled.

For details, see *32-bit and 26-bit configuration* on page A8-10.

On ARM processors which do not support 26-bit backwards-compatibility configurations, this bit reads as 1 and ignores writes.

L (bit[6])   In some ARMv3 and earlier implementations, the *abort model* of the processor could be configured:

0 = Early Abort Model selected (now obsolete)

1 = Late Abort Model selected (same as Base Updated Abort Model).

For details, see *Effects of data-aborted instructions* on page A2-17.

On later processors, this bit reads as 1 and ignores writes.

B (bit[7])   On ARM processors which support both little-endian and big-endian memory systems, this bit is used to configure the ARM processor to the endianness of the memory system:

0 = Configured for little-endian memory system

1 = Configured for big-endian memory system.

For details, see *Endianness* on page A2-23.

                   ARM DDI 0100E

On ARM processors that support only little-endian memory systems, this bit reads as 0 and ignores writes. On ARM processors that support only big-endian memory systems, this bit reads as 1 and ignores writes.

S (bit[8])   System protection bit (in an MMU-based memory system). The effect of this bit is described in *Access permissions* on page B3-16.

R (bit[9])   ROM protection bit (in an MMU-based memory system). The effect of this bit is described in *Access permissions* on page B3-16.

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 = Branch prediction disabled

1 = Branch prediction enabled.

If branch prediction cannot be disabled, this bit reads as 1 and ignores writes.

On ARM processors that do not support branch prediction, this bit reads as 0 and ignores writes.

I (bit[12])   If separate caches are used, this is the enable/disable bit for the instruction cache:

0 = Instruction cache disabled

1 = Instruction cache enabled.

If a unified cache is used or the instruction cache is not implemented, this bit reads as 0 and ignores writes. If the instruction cache cannot be disabled, this bit reads as 1 and ignores writes.

V (bit[13])   On ARM processors which support the alternative high vectors described in *High vectors* on page A2-21, this bit is used to select the location of the exception vector:

0 = Normal exception vectors selected (address range `0x00000000` - `0x0000001C`)

1 = High exception vectors selected (address range `0xFFFF0000` - `0xFFFF001C`).

On ARM processors that do not support high vectors, this bit reads as 0 and ignores writes.

RR (bit[14])   If the cache allows an alternative replacement strategy to be used which has 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).

If the cache does not allow a choice of replacement strategy this bit should ignore writes, and read as 0 or 1 according to whether the replacement strategy has a reasonably easily predictable worst-case performance.

L4 (bit[15])   For some ARM processors that support architecture version 5 or above, this bit controls a backwards-compatibility feature with previous versions of the architecture.

0 = normal behaviour for the architecture

1 = load instructions that load the PC and use bit[0] of the loaded value to determine whether to enter ARM or Thumb state (that is, to set the new value of the CPSR T bit) instead ignore bit[0] of the T bit and stay in the current execution state.

The instructions affected by this are:

- *LDM (1)* on page A4-30
- *LDR* on page A4-37
- *POP* on page A7-75

For ARM processors that support architecture versions prior to version 5, this bit should be treated as UNP/SBZP. For ARM processors that support architecture versions 5 or above, but which do not support this backwards-compatibility feature, this bit reads as zero and ignores writes.

Bits[31:16] When these bits are read, they return an UNPREDICTABLE value. When they are written, the value written must either be zero or a value previously read from the same bits.

 ARM DDI 0100E

2.5 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 and 10 are allocated to the memory protection system. See Chapter B3 *Memory Management Unit* and Chapter B4 *Protection Unit* for details of these registers.

- CP15 registers 7 and 9 are allocated to the control of caches and write buffers. See Chapter B5 *Caches and Write Buffers* for details of these registers.

- CP15 register 13 is allocated to the process ID of the Fast Context Switch Extension. See Chapter B6 *Fast Context Switch Extension* for details of this register.

- 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 11, 12 and 14 are reserved for future expansion. Accessing (reading or writing) any of these registers is UNPREDICTABLE.

Chapter B3
Memory Management Unit

This chapter describes the memory system architecture based on a *Memory Management Unit* (MMU). It contains the following sections:

3.1   About the MMU architecture

The *Memory Management Unit* (MMU) memory system architecture allows fine-grained control of a memory system. Most of the detailed control is provided through *translation tables* held in memory. Entries in these tables define the properties of memory areas of various sizes from 1KB to 1MB. The properties include:

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 is being used (see Chapter B6 *Fast Context Switch Extension*), all references to virtual addresses in this chapter mean the *modified virtual address* that it generates.

Memory access permissions

These control 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 signaled to the ARM processor.

The level of access allowed can also be affected by whether the program is running in User mode or a privileged mode, and by the use of *domains* (see *Domains* on page B3-17).

Cachability and bufferability bits (C and B)

These are described in *Cachability and bufferability* on page B5-8.

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 process of doing a full translation table lookup is known as a *translation table walk.* It is performed automatically by hardware, and has a significant execution time cost (at least one main memory access, and often two). To reduce the average cost of a memory access, the results of translation table walks are cached in one or more structures known as *Translation Lookaside Buffers* (TLBs). Usually, there is a TLB for each memory interface of the ARM implementation:

- a system with a single memory interface normally has a single unified TLB

- a system with separate instruction and data memory interfaces normally has separate instruction and data TLBs.

If the system also contains a cache or caches, the number of caches is usually determined in the same way (see *Unified or separate caches* on page B5-5). So in cached systems, there is normally one TLB per cache.

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 are likely to cease to be valid. The MMU architecture therefore supplies operations to flush TLBs.

The MMU architecture also allows specific translation table walk results to be *locked down* in a TLB. This ensures that accesses to the associated memory areas never cause a translation table walk, and has similar benefits for real-time code to those of locking down instructions and data in the cache (see *Register 9: Cache lockdown* on page B5-18).

3.2 Memory access sequence

When the ARM generates a memory access, the MMU first looks up the virtual address of the access in the TLB. If the implementation has separate data and instruction TLBs, it uses:

- the instruction TLB for an instruction fetch
- the data TLB for all other types of access.

If the TLB does not contain an entry for the virtual address, translation table walk hardware is invoked to retrieve the translation and access permission information from the translation table held in main memory. Once retrieved, the information is placed in the TLB, either in a currently unused entry or overwriting an existing entry. See *Translation process* on page B3-6 for a description of translation tables and of how the translation table walk is performed.

Once the TLB entry for a memory access is obtained, the information it contains is used as follows:

1. The C (cachable) 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 *Cachability and bufferability* on page B5-8. (If the system does not contain a cache and/or a write buffer, the relevant bits are ignored.)

2. The access permission bits and domain are used to determine whether the access is permitted. If the access is not permitted, the MMU signals a memory abort to the ARM processor. Otherwise, the access is allowed to proceed.

   The sections *Access permissions* on page B3-16, *Domains* on page B3-17 and *Aborts* on page B3-18 describe in more detail how this is done.

3. For an uncached memory access (including all memory accesses in an uncached system), the physical address is used as the address for the main memory access.

   For a cached memory access, the physical address is used as the address for the line fetch in the event of a cache miss. If a cache hit occurs, the physical address is ignored.

Figure 3-1 illustrates this for a cached system:
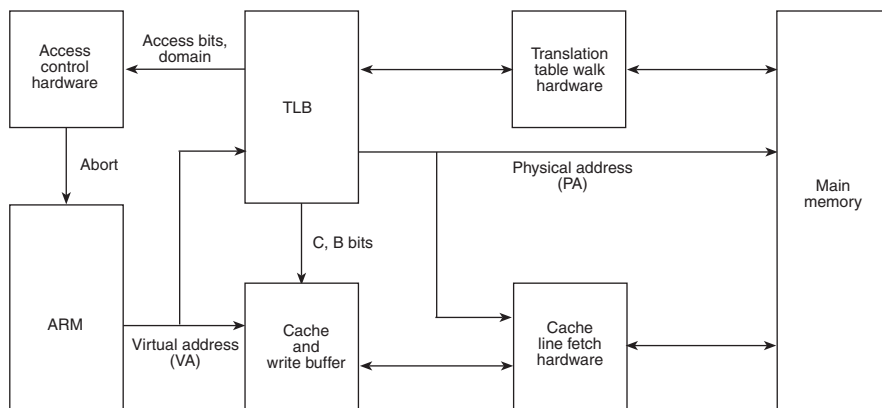
Figure 3-1 Cached MMU memory system overview

3.2.1   Enabling and disabling the MMU

The MMU can be enabled and disabled by writing bit[0] of register 1 of the System Control coprocessor. On reset, this bit is cleared to 0, so the MMU is disabled after a reset.

When the MMU is disabled, memory accesses are treated as follows:

*   It is IMPLEMENTATION DEFINED whether caches and write buffers are allowed to be enabled while the MMU is disabled.
    — For caches and write buffers which are not allowed to be enabled while the MMU is disabled, the values of the C and B bits for a memory access are irrelevant.
    — For caches and write buffers which are allowed to be enabled while the MMU is disabled, a data access is treated as uncachable and unbufferable (C == 0, B == 0). An instruction fetch is treated as uncachable (C == 0) in a system with a unified TLB, and as cachable (C == 1) in a system with a separate instruction TLB.

*   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).

Before the MMU is enabled, suitable translation tables must be set up in memory and all relevant CP15 registers must be programmed.

——— **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). It is therefore likely that any caches, for example, that are enabled at the time will need to be flushed (see *Address mapping changes* on page B5-10).

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 A2-27). It is therefore strongly recommended that code which enables or disables the MMU has identical virtual and physical addresses.

3.3 Translation process

The MMU supports memory accesses based on sections or pages:

Sections   Are comprised of 1MB blocks of memory.

Three different page sizes are supported:

Tiny pages   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. There are no similar subpages for tiny pages, so access controls for tiny pages apply to the entire page.

The translation table held in main memory has two levels:

First-level table Holds both section translations and pointers to second-level tables.

Second-level tables Hold both large and small page translations. One type of second-level table can also hold tiny page translations.

The MMU translates virtual addresses generated by the CPU into physical addresses to access external memory, and also derives and checks the access permission. There are four routes by which the address translation (and therefore permission check) takes place. The route taken depends on whether the address in question has been marked as a section-mapped access or a page-mapped access. A page-mapped access can be a large, small, or tiny page access.

However, the translation process always starts out in the same way, as described below, with a first-level fetch. A section-mapped access only requires a first level fetch, but a page-mapped access also requires a second-level fetch.

3.3.1 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 (CP15 register 2) holds the physical address of the base of the first-level table. 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.

3.3.2 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 illustrated in Figure 3-2. 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.
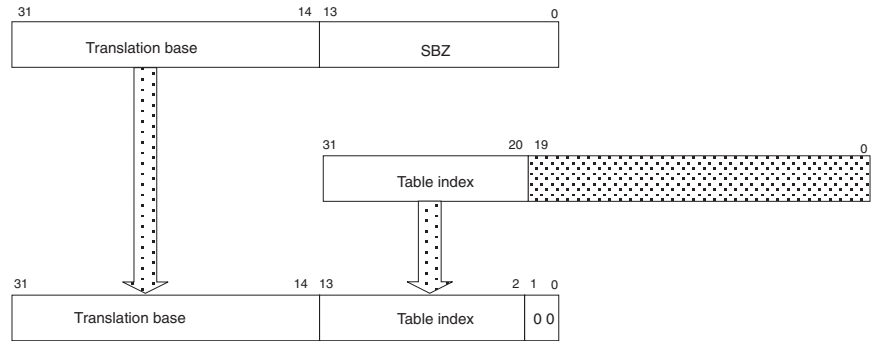
Figure 3-2 Accessing the translation table first-level descriptors

3.3.3 First-level descriptors

Each entry in the first-level table is a descriptor of how its associated 1MB virtual address range is mapped. There are the following possibilities, selected by bits[1:0] of the descriptor as shown in Table 3-1:

- If bits[1:0] == 0b00, the associated virtual addresses are unmapped, and attempts to access them generate a translation fault (see *Aborts* on page B3-18). 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 *Section descriptor and translating section references* on page B3-8 for details of how it is interpreted.

- If bit[0] == 1, the entry gives the physical address of a coarse second-level table (if bit[1] == 0) or of a fine second-level table (if bit[1] == 1). Each type of table specifies how the associated 1MB virtual address range is mapped. Coarse tables are smaller, at 1KB per table (compared with 4KB per table for fine tables). However, coarse tables can only map large pages and small pages, while fine tables can map large, small and tiny pages. The two types of table are described in *Coarse page table descriptor* on page B3-9 and *Fine page table descriptor* on page B3-10.

Table 3-1 First-level descriptor format

	31 ... 20	19 ... 12	11 10	9	8 ... 5	4 3 2	1	0
Fault	IGN						0	0
Coarse page table	Coarse page table base address			SBZ	Domain	IMP	0	1
Section	Section base address	SBZ	AP	SBZ	Domain	IMP C B	1	0
Fine page table	Fine page table base address		SBZ		Domain	IMP	1	1

3.3.4 Section descriptor and translating section references

If the first-level descriptor is a section descriptor, the fields have the following meanings:

Bits[1:0]   Identify the type of descriptor (0b10 marks a section descriptor).

Bits[3:2]   The cachable and bufferable bits (see *Cachability and bufferability* on page B5-8).

Bit[4]   The meaning of this bit is IMPLEMENTATION DEFINED.

Bits[8:5]   The domain field specifies one of the sixteen possible domains for the section controlled by this descriptor.

Bit[9]   This bit is not currently used, and should be zero.

Bits[11:10]   Access permissions. These bits control the access to the section. See Table 3-3 on page B3-16 for the interpretation of these bits.

Bits[19:12]   These bits are not currently used, and should be zero.

Bits[31:20]   The Section Base Address forms the top 12 bits of the physical address.

Figure 3-3 illustrates the complete section translation sequence.

—— **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 B3-16.
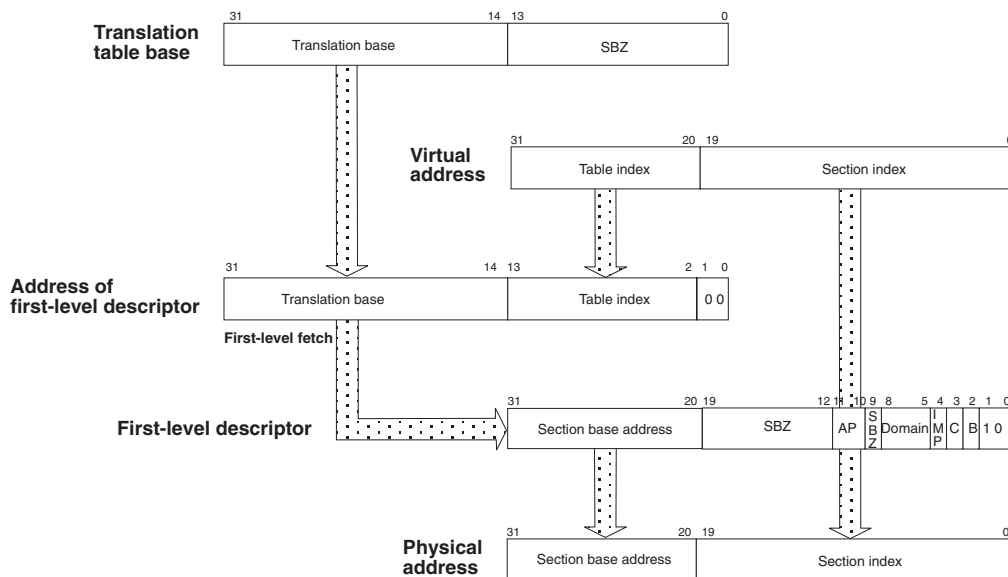
Figure 3-3 Section translation

       ARM DDI 0100E

3.3.5 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.

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 is not currently used, and should be zero.

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 3-4.
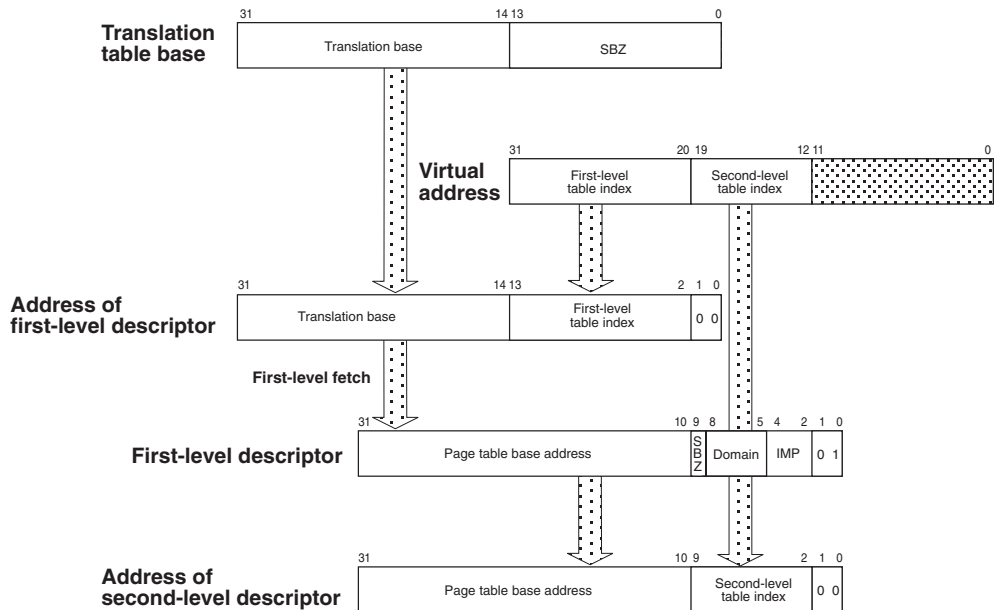
Figure 3-4 Accessing coarse page table second-level descriptors

3.3.6   Fine page table descriptor

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 3-5.
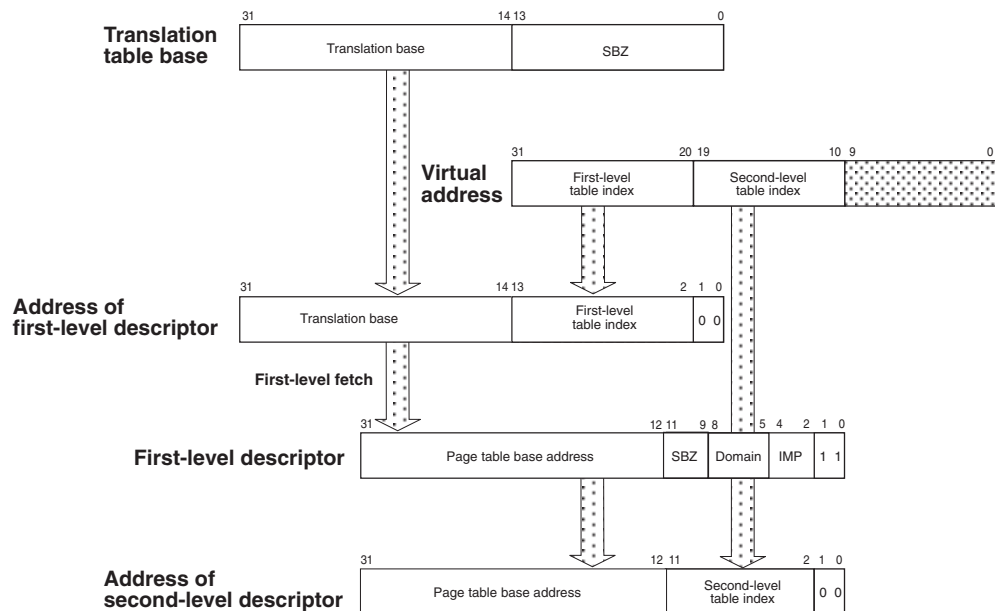
Figure 3-5 Accessing fine page table second-level descriptors

Copyright © 1996-2000 ARM Limited. All rights reserved.   ARM DDI 0100E

3.3.7 Second-level descriptor

Each entry in a coarse second-level table corresponds to how a 4KB virtual address range is mapped, and each entry in a fine second-level table corresponds to how a 1KB virtual address range is mapped. The entries are page descriptors, and can describe a page which is larger than 4KB or 1KB respectively. When this happens, the descriptor must be repeated enough times in the second-level table to ensure that the same descriptor is always used for the page, regardless of which virtual address is being accessed within the page.

There are four possibilities for a second-level descriptor, selected by bits[1:0] of the descriptor as shown in Table 3-2:

* If bits[1:0] == 0b00, the associated virtual addresses are unmapped, and attempts to access them generate a translation fault (see *Aborts* on page B3-18). 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] == 0b01, the entry is a large page descriptor, describing 64KB of virtual addresses. See *Translating large page references* on page B3-13 for details.

    A large page descriptor must be repeated 64 times in a fine second-level table or 16 times in a coarse second-level table to ensure that all of its virtual addresses are described.

* If bits[1:0] == 0b10, the entry is a small page descriptor, describing 4KB of virtual addresses. See *Translating small page references* on page B3-14 for details.

    A small page descriptor must be repeated four times in a fine second-level table to ensure that all of its virtual addresses are described. In a coarse second-level table, only one instance of each small page descriptor is needed.

* If bits[1:0] == 0b11, the entry is a tiny page descriptor, describing 1KB of virtual addresses. See *Translating tiny page references* on page B3-15 for details.

    Only one instance of each tiny page descriptor is needed in a fine second-level table. Tiny page descriptors must not appear in coarse second-level tables. If they do, the results are UNPREDICTABLE.

Table 3-2 Second-level descriptor format

	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

ARM DDI 0100E   *Copyright © 1996-2000 ARM Limited. All rights reserved.*   B3-11

Large page descriptor fields

The fields in a large page descriptor have the following meanings:

Bits[1:0]   Identify the type of descriptor.

Bits[3:2]   Are the cachable and bufferable bits (see *Cachability and bufferability* on page B5-8).

Bits[11:4]   Are the access permission bits. These bits control access to the page. See Table 3-3 on page B3-16 for the interpretation of these bits. Large pages are split into four subpages:

    AP0   Encodes the access permissions for the first subpages.

    AP1   Encodes the access permissions for the second subpages.

    AP2   Encodes the access permissions for the third subpages.

    AP3   Encodes the access permissions for the fourth (last) subpages.

Bits[15:12]   Are not currently used, and must be zero.

Bits[31:16]   Are used to form the corresponding bits of the physical address.

Small page descriptor fields

The fields in a small page descriptor have the following meanings:

Bits[1:0]   Identify the type of descriptor.

Bits[3:2]   Are the cachable and bufferable bits (see *Cachability and bufferability* on page B5-8).

Bits[11:4]   Are the access permission bits. These bits control access to the page. See Table 3-3 on page B3-16 for the interpretation of these bits. Small pages are split into four subpages:

    AP0   Encodes the access permissions for the first subpages.

    AP1   Encodes the access permissions for the second subpages.

    AP2   Encodes the access permissions for the third subpages.

    AP3   Encodes the access permissions for the fourth (last) subpages.

Bits[31:12]   Are used to form the corresponding bits of the physical address.

Tiny page descriptor fields

The fields in a tiny page descriptor have the following meanings:

Bits[1:0]   Identify the type of descriptor.

Bits[3:2]   Are the cachable and bufferable bits (see *Cachability and bufferability* on page B5-8).

Bits[5:4]   Are the access permission bits. See Table 3-3 on page B3-16 for the interpretation of these bits, which apply to the entire tiny page.

Bits[9:6]   Are not currently used, and must be zero.

Bits[31:10]   Are used to form the corresponding bits of the physical address.

 ARM DDI 0100E

3.3.8 Translating large page references

Figure 3-6 shows the complete translation sequence for a 64KB large page in a coarse second-level table. The translation sequence for a large page in a fine second-level table is similar, but with the address of the second-level descriptor being determined as shown in *Fine page table descriptor* on page B3-10.

——— **Note** ———

As 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. Similarly, each page table entry for a large page must be repeated 64 times in a fine page table.
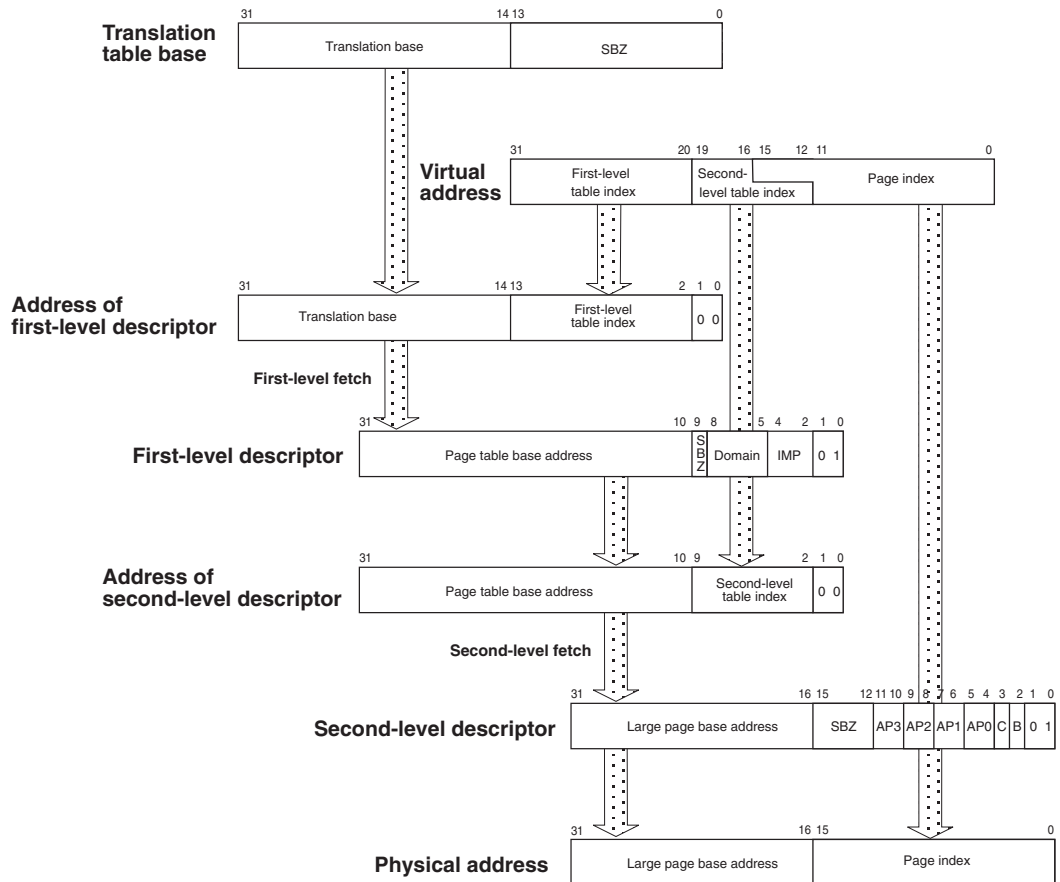
Figure 3-6 Large page translation in a coarse second-level table

Copyright © 1996-2000 ARM Limited. All rights reserved.

3.3.9 Translating small page references

Figure 3-7 shows the complete translation sequence for a 4KB small page in a coarse second-level table. The translation sequence for a small page in a fine second-level table is similar, but with the address of the second-level descriptor being determined as shown in *Fine page table descriptor* on page B3-10.

—— **Note** ——

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. So each page table entry for a small page must be repeated four times (in consecutive memory locations) in a fine page table.
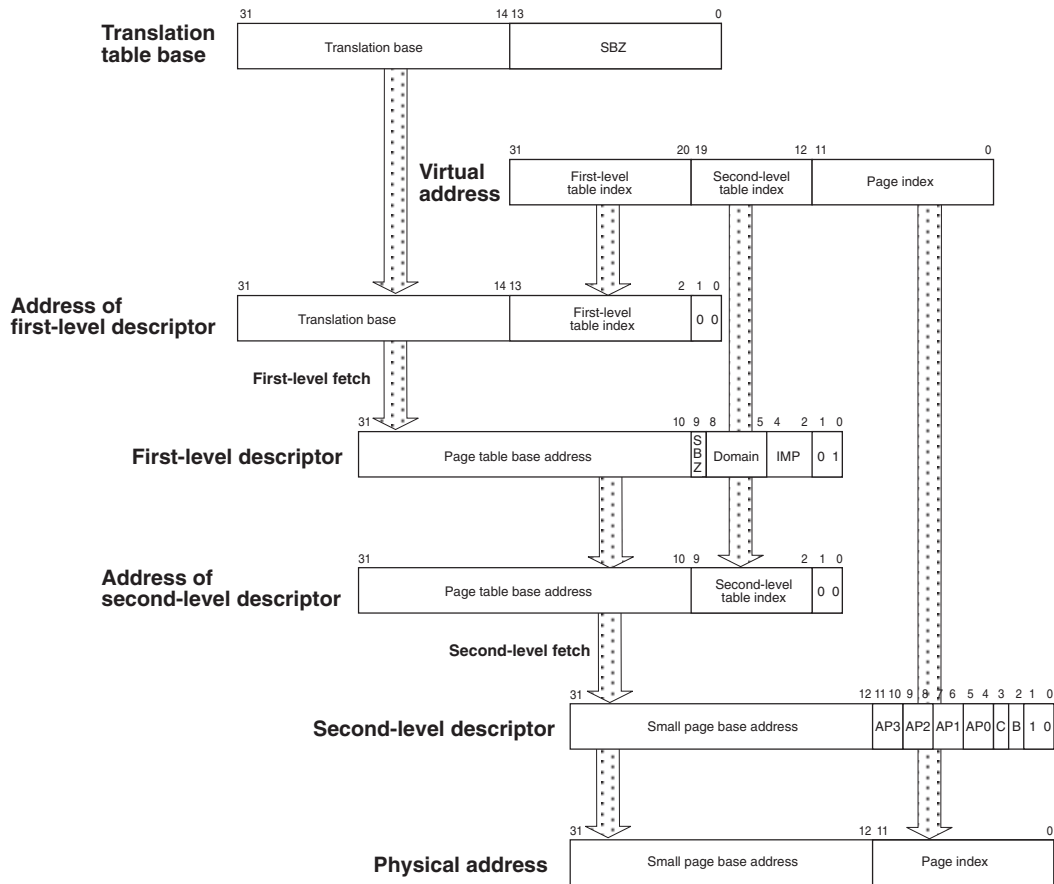
Figure 3-7 Small page translation in a coarse second-level table

3.3.10 Translating tiny page references

Figure 3-8 shows the complete translation sequence for a 1KB tiny page in a fine second-level table.

— **Note** —

Tiny pages can not appear in a coarse second-level table.
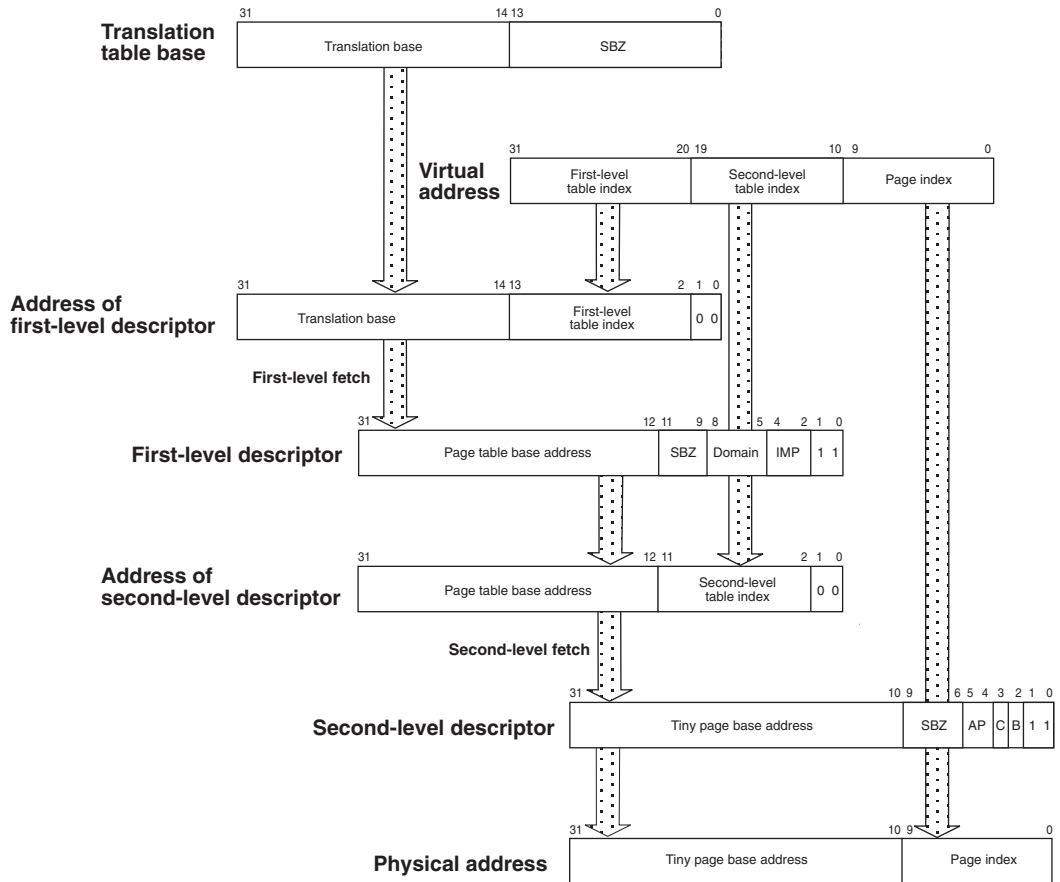
Figure 3-8 Tiny page translation in a fine second-level table

3.4   Access permissions

The access permission bits in section and page descriptors control access to the corresponding section or page. The access permissions are modified by the System (S) and ROM (R) control bits in CP15 register 1. Table 3-3 describes the meaning of the access permission bits in conjunction with the S and R bits. If an access is made to an area of memory without the required permission, a Permission Fault is raised (see *Aborts* on page B3-18).

Table 3-3 MMU access permissions

AP	S	R	Privileged permissions	User permissions
0b00	0	0	No access	No access
0b00	1	0	Read only	No access
0b00	0	1	Read only	Read only
0b00	1	1	UNPREDICTABLE	UNPREDICTABLE
0b01	x	x	Read/write	No access
0b10	x	x	Read/write	Read only
0b11	x	x	Read/write	Read/write

       ARM DDI 0100E

3.5 Domains

A domain is a collection of sections, large pages and small pages. The ARM architecture supports 16 domains. Access to each domain is controlled by a 2-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, access data), and are guarded by the access permissions of the individual sections and pages that make up the 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 of individual sections and pages in the 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 3-4 illustrates the encoding of the bits in the Domain Access Control Register.

Table 3-4 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 section or page descriptor
0b10	Reserved	Using this value has UNPREDICTABLE results
0b11	Manager	Accesses are not checked against the access permission bits in the section or page descriptor, so a permission fault cannot be generated

3.6 Aborts

Mechanisms that can cause the ARM processor to halt execution because of memory access restrictions are:

MMU fault   The MMU detects the restriction and signals the processor.

External abort   The external memory system signals an illegal memory access.

Collectively, MMU faults and external aborts are just called *aborts*. Accesses that cause aborts are said to be aborted.

If the memory request that is aborted is an instruction fetch, then a Prefetch Abort exception is raised if and when the processor attempts to execute the instruction corresponding to the illegal access. If the aborted access is a data access, a Data Abort exception is raised. See *Exceptions* on page A2-13 for more information about Prefetch and Data Aborts.

3.6.1 MMU faults

The MMU generates four types of fault:

- alignment fault
- translation fault
- domain fault
- permission fault.

The memory system can abort three types of access:

- line fetches
- memory accesses (uncached or unbuffered accesses)
- translation table accesses.

Aborts that are detected by the MMU are stopped before any external memory access takes place. It is the responsibility of the external system to stop external accesses that cause external aborts.

The System Control coprocessor contains two registers which are updated when a data access is aborted. These registers are not updated for Prefetch Aborts, as the aborted instruction might not be executed due to changes in program flow.

Fault Address Register (FAR) and Fault Status Register (FSR)

Aborts resulting from data accesses (Data Aborts) are immediately acted upon by the CPU. The Fault Status Register (FSR) is updated with a 4-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 Fault Address Register (FAR). If a data access simultaneously generates more than one type of Data Abort, they are prioritized in the order given in Table 3-5 on page B3-19.

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 will never update 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.

Table 3-5 Priority encoding of fault status

Priority	Sources		FS [3:0]	Domain [3:0]	FAR
Highest	Terminal Exception		0b0010	Invalid	IMPLEMENTATION DEFINED
	Vector Exception		0b0000	Invalid	Valid
	Alignment		0b00x1	Invalid	Valid
	External Abort on Translation	First level	0b1100	Invalid	Valid
		Second level	0b1110	Valid	Valid
	Translation	Section	0b0101	Invalid	Valid
		Page	0b0111	Valid	Valid
	Domain	Section	0b1001	Valid	Valid
		Page	0b1011	Valid	Valid
	Permission	Section	0b1101	Valid	Valid
		Page	0b1111	Valid	Valid
	External Abort on Linefetch	Section	0b0100	Valid	Valid
		Page	0b0110	Valid	Valid
Lowest	External Abort on Non-linefetch	Section	0b1000	Valid	Valid
		Page	0b1010	Valid	Valid

───── **Note** ─────

Alignment faults can write either 0b0001 or 0b0011 into FS[3:0].

Invalid values in Domain[3:0] occur because the fault is raised before a valid domain field has been loaded.

Any abort masked by the priority encoding can be regenerated by fixing the primary abort and restarting the instruction.

The FS[3:0] encoding for Vector Exception breaks from the pattern that FS[0] is zero for all external aborts.

Fault-checking sequence

The sequence by which the MMU checks for access faults is slightly different for Sections and Pages. Figure 3-9 illustrates the sequence for both types of access.

Figure 3-9 Sequence for checking faults

Terminal exception

This indicates that an irrecoverable fault has occurred. The circumstances under which this can happen (if at all) are IMPLEMENTATION DEFINED.

Vector exception

When the processor is in a 32-bit configuration (**PROG32** is active) and in a 26-bit mode (CPSR[4] == 0), data access (but not instruction fetches) to the normal exception vectors (address `0x0` to `0x1F`) cause a Data Abort, known as a vector exception. It is IMPLEMENTATION DEFINED if vector exceptions are generated when the MMU is not enabled. For more information, see *Vector exceptions* on page A8-11.

Alignment fault

If alignment faults are enabled, an alignment fault is generated on any data word access whose address is not word-aligned (virtual address bits [1:0] != 0b00), or any halfword access that is not halfword-aligned (virtual address bit[0] != 0). Alignment faults are not generated on any instruction fetch, or on any byte access.

——— **Note** ———

If the access generates an alignment fault, it is aborted without reference to further permission checks. It is IMPLEMENTATION DEFINED if alignment exceptions are generated when the MMU is not enabled.

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.

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
* Page.

In both cases, the first descriptor holds the 4-bit Domain field which selects one of the sixteen 2-bit domains in the Domain Access Control Register. The two bits of the specified domain are then checked for access permissions as detailed in Table 3-4 on page B3-17:
* In the case of a section, the domain is checked when the first-level descriptor is returned.
* In the case of a page, the domain is checked when the second-level descriptor is returned.

If the specified access is marked as No Access in the Domain Access Control Register, either a Section Domain Fault or Page Domain Fault occurs.

Permission fault

There are section permission faults and subpage permission faults.

Permission faults are checked at the same time as domain faults. If the 2-bit domain field returns client (01), the permission access check is invoked as follows:

Section   If the first-level descriptor defines a section access, the AP bits of the descriptor define whether or not the access is allowed, according to Table 3-3 on page B3-16. If the access is not allowed, a section permission fault is generated.

Subpages   If the first-level descriptor defines a page-mapped access, the second-level descriptor can define a tiny page, a small page or a large page. If the second-level descriptor defines a tiny page, it contains a single access permission field (AP), which applies to the whole page. Otherwise, the second-level descriptor contains four access permission fields (AP3, AP2, AP1, AP0), each corresponding to one quarter of the page.

For small pages, AP3 is selected by the top 1KB of the page, and AP0 is selected by the bottom 1KB of the page. For large pages, AP3 is selected by the top 16KB of the page, and AP0 is selected by the bottom 16KB of the page. The selected AP bits are then interpreted in exactly the same way as for a section, (see Table 3-3 on page B3-16) the only difference being that the fault generated is a subpage permission fault.

3.6.2   External aborts

In addition to the MMU faults, the ARM architecture defines an external abort pin which can be used to flag an error on an external memory access. However, not all accesses can be aborted in this way, so this pin must be used with great care. The following accesses can be externally aborted and restarted safely:

*   reads
*   unbuffered writes
*   first-level descriptor fetch
*   second-level descriptor fetch
*   semaphores in uncachable/unbufferable memory areas.

A linefetch can be safely aborted on any word in the line transfer. If the abort happens on data that has been requested by the processor (rather than data that is being fetched as the remainder of a cache line), the access is aborted. Any data transferred that is not immediately accessed (the remainder of the cache line) is only aborted when it is accessed.

It is IMPLEMENTATION DEFINED if the FAR points to the start address of the cache line, or the address that generated the abort.

Buffered writes cannot be externally aborted. Therefore, the system must be configured so that it does not do buffered writes to areas of memory which are capable of flagging an external abort, or a different mechanism needs to be used to signal the abort (an interrupt for example).

The value of a memory location that causes an abort is UNPREDICTABLE after the abort.

     ARM DDI 0100E

3.7   CP15 registers

The MMU is controlled with the System Control coprocessor registers 2, 3, 4, 5, 6, 8 and 10, and some bits of register 1.

3.7.1   MMU control bits in register 1

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.

On systems without an MMU or Protection Unit, this bit must read as zero and ignore writes.

A (bit[1])   This bit enables and disables alignment fault checking (see *Alignment fault* on page B3-21):

0 = Alignment fault checking disabled

1 = Alignment fault checking enabled.

S (bit[8])   This is the system protection bit. The effect of this bit is described in *Access permissions* on page B3-16.

R (bit[9])   This is the ROM protection bit. The effect of this bit is described in *Access permissions* on page B3-16.

3.7.2   Register 2: Translation table base

31	14 13	0
Translation Table Base		UNP/SBZP
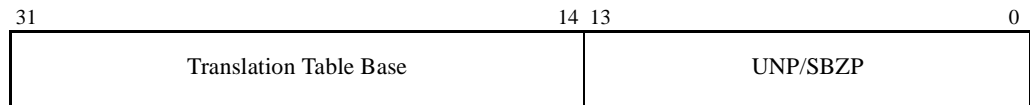
Reading from CP15 register 2 returns the physical address of the currently active first-level translation table in bits[31:14] and an UNPREDICTABLE value in bits[13:0]. The CRm and opcode2 fields are ignored when reading CP15 register 2 and should be zero.

Writing to CP15 register 2 updates the physical address of the currently active first-level translation table from the value in bits[31:14] of the written value. Bits[13:0] must be written as zero or as a value previously read from bits[13:0] of this register. The CRm and opcode2 fields are ignored when writing CP15 register 2 and should be zero.

3.7.3   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

Reading from CP15 register 3 returns the value of the Domain Access Control Register. The CRm and opcode2 fields are ignored when reading CP15 register 3 and should be zero.

Writing to CP15 register 3 writes the value of the Domain Access Control Register. The CRm and opcode2 fields are ignored when writing CP15 register 3 and should be zero.

The Domain Access Control Register consists of sixteen 2-bit fields (see *Domains* on page B3-17), each defining the access permissions for one of the sixteen Domains (D15–D0).

3.7.4   Register 4: Reserved

Reading and writing CP15 register 4 is UNPREDICTABLE.

3.7.5   Register 5: Fault status

31 9	8	7 4	3 0
UNP/SBZP	0	Domain	Status

Reading CP15 register 5 returns the value of the Fault Status Register (FSR). The FSR contains the source of the last data fault. Only the bottom nine bits are returned. The upper 23 bits are UNPREDICTABLE. The FSR indicates the domain and type of access being attempted when an abort occurred.

Bit[8]   Returns zero.

Bits[7:4]   Specify which of the sixteen domains (D15–D0) was being accessed when a fault occurred.

Bits[3:0]   Indicate the type of access being attempted. The encoding of these bits is shown in Table 3-5 on page B3-19.

The FSR is updated for data faults. It is IMPLEMENTATION DEFINED whether the FSR is updated for prefetch faults (for details, see *Fault Address Register (FAR) and Fault Status Register (FSR)* on page B3-18). The CRm and opcode2 fields are ignored when reading CP15 register 5 and should be zero.

Writing CP15 register 5 sets the Fault Status Register to the value of the data written. This is useful for a debugger to restore the value of the FSR. The upper 24 bits written should be zero or a value previously read from the FSR. The CRm and opcode2 fields are ignored when writing CP15 register 5 and should be zero.

                    ARM DDI 0100E

3.7.6 Register 6: Fault address

31	0
Fault address	

Reading CP15 register 6 returns the value of the FAR. The FAR holds the virtual address of the access which was attempted when a fault occurred. The FAR is updated for data faults. It is IMPLEMENTATION DEFINED whether the FAR is updated for prefetch faults (for details see *Fault Address Register (FAR) and Fault Status Register (FSR)* on page B3-18). The CRm and opcode2 fields are ignored when reading CP15 register 6 and should be zero.

Writing CP15 register 6 sets the Fault Address Register to the value of the data written. This is useful for a debugger to restore the value of the FAR. The CRm and opcode2 fields are ignored when writing CP15 register 6 and should be zero.

——— **Note** ———

If the *Fast Context Switch Extension* (FCSE) described in Chapter B6 is being used, then:

- When the FAR is updated for a memory fault, the *modified virtual address* is written to the FAR.

- When the FAR is read by an MRC instruction or written by an MCR instruction, its value is being treated as data and so no address modification is performed by the FCSE.

3.7.7 Register 8: TLB functions

CP15 register 8 is a write-only register which is used to control TLBs. Table 3-6 on page B3-26 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.

If any of the following operations are used on an implementation with a unified TLB, the equivalent function is performed on the unified TLB:
- invalidate instruction TLB
- invalidate instruction single entry
- invalidate entire data TLB
- invalidate data single entry.

Otherwise, if a function not relevant to a particular implementation is performed, the results are UNPREDICTABLE.

Attempting to read CP15 register 8 with an MRC instruction is UNPREDICTABLE.

The invalidate single entry operations can be used to improve performance on some implementations when only a small amount of memory is being remapped. For each remapped memory area (section, small page or large page), the invalidate single entry operation needs to be performed on a virtual address in the memory area. The performance improvement comes from not having to subsequently reload TLB entries relating to memory areas that were not remapped.

——— **Note** ———

The performance improvement for the invalidate single entry operations is not guaranteed. Implementations are free to invalidate more than the single requested entry, up to and including invalidating the entire TLB.

——— **Caution** ———

When memory is remapped, all TLB entries relating to the old mapping must be invalidated. If this is not done, it is possible to get into a state where two TLB entries relate to overlapping ranges of virtual addresses. Any attempt to address the overlapped virtual memory addresses results in UNPREDICTABLE behavior at best, and might even physically damage the MMU on some implementations.

It is therefore strongly recommended that great care is taken to invalidate the TLB appropriately whenever memory is remapped.

Table 3-6 TLB functions

Function	\<opcode2\>	\<CRm\>	Data	Instruction
Invalidate entire unified TLB or both instruction and data TLBs	0b000	0b0111	SBZ	`MCR p15, 0, Rd, c8, c7, 0`
Invalidate unified single entry	0b001	0b0111	Virtual address	`MCR p15, 0, Rd, c8, c7, 1`
Invalidate entire instruction TLB	0b000	0b0101	SBZ	`MCR p15, 0, Rd, c8, c5, 0`
Invalidate instruction single entry	0b001	0b0101	Virtual address	`MCR p15, 0, Rd, c8, c5, 1`
Invalidate entire data TLB	0b000	0b0110	SBZ	`MCR p15, 0, Rd, c8, c6, 0`
Invalidate data single entry	0b001	0b0110	Virtual address	`MCR p15, 0, Rd, c8, c6, 1`

——— **Note** ———

If the *Fast Context Switch Extension* (FCSE) described in Chapter B6 is being used, the virtual addresses passed to CP15 by some of the functions in Table 3-6 are passed as data. This means that no address modification is performed for them by the FCSE.

3.7.8 Register 10: TLB lockdown

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>` should be zero.

`<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 functions* on page B5-15 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 you are using the *Fast Context Switch Extension* (FCSE) (see Chapter B6), take care 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 minimise 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 in order 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.

ARM DDI 0100E

Chapter B4
Protection Unit

This chapter describes the ARM Protection Unit. It contains the following sections:

- *About the Protection Unit* on page B4-2
- *Overlapping regions* on page B4-5
- *CP15 registers* on page B4-6.

4.1 About the Protection Unit

The *Protection Unit* memory system architecture provides a considerably simpler alternative to the MMU memory system architecture described in Chapter B3 *Memory Management Unit*. 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 Fast Context Switch Extension is used, as described in Chapter B6 *Fast Context Switch Extension*, the physical address is the same as the *modified virtual address* that it generates).

4.1.1 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 cachable 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 B4-5.

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 needs to 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.

4.1.2 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 (cachable) 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 *Cachability and bufferability* on page B5-8. (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 B4-7 and *Exceptions* on page A2-13 describe this in more detail.

Figure 4-1 illustrates the memory access sequence for a cached system:
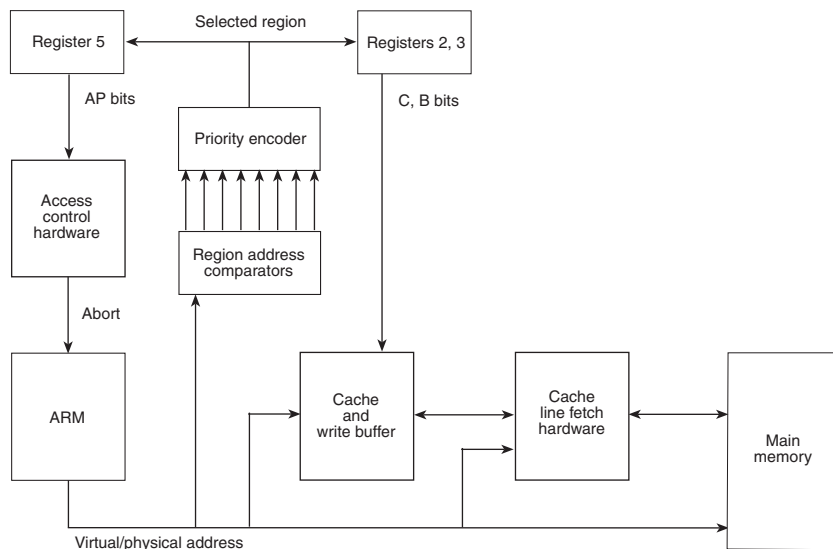
Figure 4-1 Cached Protection Unit memory system overview

Copyright © 1996-2000 ARM Limited. All rights reserved.

4.1.3 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 uncachable, unbufferable and non-aborting.

 ARM DDI 0100E

4.2   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 4-2. Because there is a clash, the attributes associated with region 2 are applied. In this case, the load would not abort.
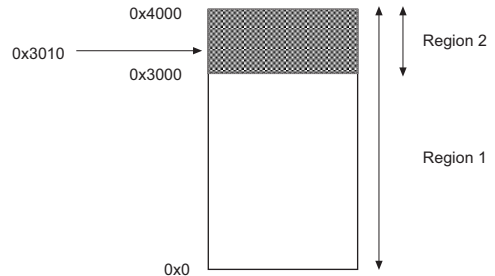
Figure 4-2 Overlapping memory regions

4.2.1   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.

4.3 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.

4.3.1 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 B3-21. If it does, the A bit (bit[1]) of register 1 behaves as described there and in *MMU control bits in register 1* on page B3-23. Otherwise, the A bit should be zero.

4.3.2 Register 2: Cachability bits

31	8	7	6	5	4	3	2	1	0
UNP/SBZP		C7	C6	C5	C4	C3	C2	C1	C0
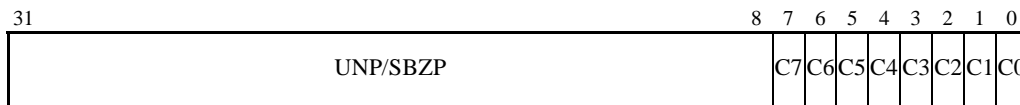
Reading from CP15 register 2 returns the cachable (C) bits for the eight protection regions in bits[7:0], with bit[n] corresponding to region n, and an UNPREDICTABLE value in bits[31:8].

Writing to CP15 register 2 updates the cachable (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 *Cachability and bufferability* on page B5-8.

4.3.3 Register 3: Bufferability bits

31	8	7	6	5	4	3	2	1	0
UNP/SBZP		B7	B6	B5	B4	B3	B2	B1	B0
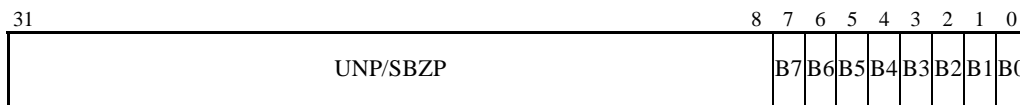
Reading from CP15 register 3 returns the bufferable (B) bits for the eight protection regions in bits[7:0], with bit[n] corresponding to region n, and an UNPREDICTABLE value in bits[31:8].

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 *Cachability and bufferability* on page B5-8.

4.3.4 Registers 4, 8, 10: Reserved

Accessing (reading or writing) any of these registers is UNPREDICTABLE.

4.3.5 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 4-1 on page B4-7. If the requested type of access is not permitted, an abort is signaled to the ARM processor.

Table 4-1 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

4.3.6 Register 6: Protection area control

31	12	11	6	5	1	0
Base address		UNP/SBZP		Size		E
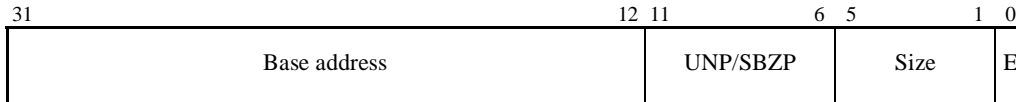
Reading from CP15 register 6 returns the current base address, size and enabled/disabled status of a protection region, in the format shown in the above diagram. The value read for bits[11:6] is UNPREDICTABLE.

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 4-2 on page B4-9.

- 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 may also be constrained to be zero. These additional constraints are shown in Table 4-2 on page B4-9. If they are broken, the protection region is misaligned, and UNPREDICTABLE behavior might result.

Table 4-2 Region size encoding

Size field	Area size	Base area constraints
0b00000 to 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 B5
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:

5.1 About caches and write buffers

Caches and write buffers can be used in ARM memory systems to improve their average performance.

A *cache* is a block of high-speed memory locations whose addresses can be changed, and whose purpose is to increase the average speed of a memory access. Each memory location of a cache is known as a *cache line*.

Normally, changes to the address of a cache line occur automatically. Whenever the processor loads data from a memory address and no cache line currently holds that data, a cache line is allocated to that address and the data is read into the cache line. If data at the same address is accessed again before the cache line is re-allocated to another address, the cache can process the memory access at high speed. So a cache typically speeds up the second and subsequent accesses to the data. In practice, these second and subsequent accesses are common enough for this is to produce a significant performance gain. This effect is known as *temporal locality*.

To reduce the percentage overhead of storing the current addresses of the cache lines, each cache line normally consists of several memory words. This increases the cost of the first access to a cache line, since several words need to be loaded from main memory to satisfy a request for just one word. However, it also means that a subsequent access to another word in the same cache line can be processed by the cache at high speed. This sort of access is also common enough to increase performance significantly. This effect is known as *spatial locality*.

A memory access which can be processed at high speed because the data it addresses is already in the cache is known as a *cache hit*. Other memory accesses are called *cache misses*.

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 (such as data size) are written to the write buffer at high speed. The write buffer then completes the store at main memory speed, which 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 due to:

- 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 these problems, and describes cache and write buffer control facilities that can be used to work around them. They are common to the Memory Management Unit system architecture described in Chapter B3 *Memory Management Unit* and the Protection Unit system architecture described in Chapter B4 *Protection Unit*.

———— **Note** ————

The caches described in this chapter are accessed using the virtual address of the memory access. This implies that they will need to be invalidated and/or cleaned when the virtual-to-physical address mapping changes or in certain other circumstances, as described in *Memory coherency* on page B5-10.

If the *Fast Context Switch Extension* (FCSE) described in Chapter B6 is being used, all references to virtual addresses in this chapter mean the *modified virtual address* that it generates.

5.2   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 this block (and therefore the length of the 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. Such blocks of main memory locations are called *memory cache lines* or (loosely) just *cache lines*.

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 virtual address supplied by the ARM processor match the same bits of the virtual address associated with a valid cache line.

To simplify and speed up the process of determining whether a cache hit occurs, a cache is usually divided into a number of *cache sets*. The number of cache sets 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.

The remaining bits of the virtual address (bits[31:L+S]) are known as its *tag bits*. 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.

Figure 5-1 illustrates how the virtual address is used to look up data or instructions in the cache.
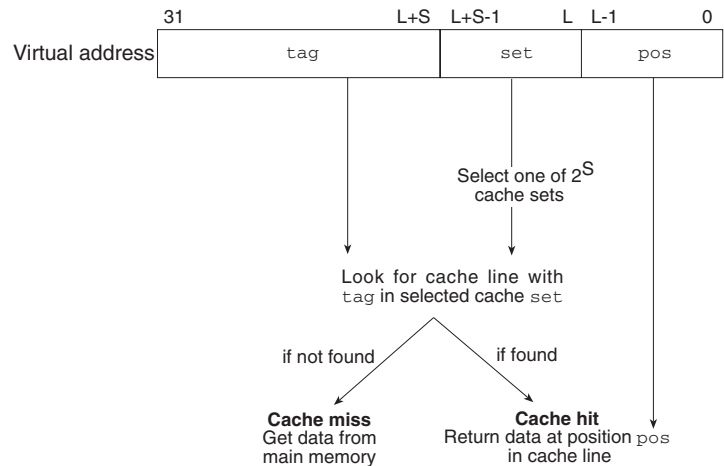
Figure 5-1 Cache look-up

5.2.1 Set-associativity

The *set-associativity* of a cache is the number of cache lines in each of its cache sets. 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, which 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 needs to 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 just needs to select and check 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 *index*. Some cache operations take a cache line index as a parameter, to allow a software loop to work systematically through a cache set.

5.2.2 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 × LINELEN.

- The number NSETS of cache sets making up the cache.

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.

If the System Control coprocessor supports the Cache Type register, it can be used to determine these cache size parameters (see *Cache Type register* on page B2-9).

5.3   Types of cache

There are many different possible types of cache, which can be distinguished by implementation choices such as:

*   how big they are
*   how they handle instruction fetches
*   how they handle data writes
*   how much of the cache is eligible to hold any particular item of data.

A number of these implementation choices are detailed in the subsections below. Also see *Cache Type register* on page B2-9 for details of how most of these choices can be determined for implementations which include a Cache Type register.

―――― **Note** ――――

A high-performance memory system can contain more than one level of cache, with the first level being small and very high speed, the next level being bigger and slower, and so on, out to main memory, which is the largest and slowest component of the memory system. Furthermore, different cache levels can be of different types.

This chapter only describes the first (or only) level of cache, as does the Cache Type register. If a memory system implementation provides facilities to control second or higher level caches, details of those facilities are IMPLEMENTATION DEFINED.

Accordingly, all references to *main memory* in the rest of this chapter refer to all of the memory system beyond the first level cache, including any further levels of cache.

5.3.1   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*.

Alternatively, a memory system can use a different cache to process instruction fetches to 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* respectively.

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* on page B5-10).

It is also possible for a memory system to have an instruction cache but no data cache, or vice versa. 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.

5.3.2 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. As this cache line will eventually be re-allocated to another address, the main memory location for the data also needs to have the new value written to it. 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*, which means that it contains data values which 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.

  Another common term for a write-back cache is a *copy-back cache*.

The main disadvantage of write-through caches is that if the processor speed becomes high enough relative to that of main memory, it generates data stores faster than they can be processed by the write buffer. The result is that the processor is slowed down by having to wait for the write buffer to be able to accept more data.

Because a write-back cache only stores to main memory once when a cache line is re-allocated, even if many stores have occurred to the cache line, write-back caches normally generate fewer stores to main memory than write-through caches. This 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* on page B5-10)

- a longer worst-case sequence of main memory operations before a data load can be completed, which can increase the system's worst-case interrupt latency

- increased complexity of implementation.

Some write-back caches allow a choice to be made between write-back and write-through behavior (see *Cachability and bufferability* on page B5-8).

5.3.3 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.

Whether write-allocate or read-allocate caches are used in an ARM memory system is IMPLEMENTATION DEFINED.

5.3.4 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 in which 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 which 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.

Some caches allow a choice of the replacement strategy in use. Typically, one choice is a simple, easily predictable strategy like round-robin replacement, which 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.

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.

When a cache allows a choice of replacement strategies, the second choice is normally a strategy like random replacement which 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.

5.4 Cachability 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 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 → 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. But if a write buffer or write-back cache delays the store, the I/O device might 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 Memory Management Unit and the Protection Unit architectures allow a memory area to be designated as uncachable, unbufferable or both. This is done by using the memory address to generate two bits (C and B) for each memory access. Details of how the C and B bits are produced for each architecture can be found in Chapter B3 *Memory Management Unit* and Chapter B4 *Protection Unit*.

Table 5-1 shows how the C and B bits are interpreted for write-through caches, write-back caches without selectable write-through behavior, and write-back caches with selectable write-through behavior.

Table 5-1 Interpretation of Cachable 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

 ARM DDI 0100E

The purpose of making a memory area unbufferable is to prevent stores to it being delayed. However, if the area is cachable 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 in Table 5-1 on page B5-8.

——— **Note** ———

- The reason that a memory-mapped I/O location generally needs to be marked as uncachable 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 also needs to be told not to optimize away these loads and stores. 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 uncachable 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 uncachable 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.

5.5 Memory 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.

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.

The following subsections discuss particular aspects of memory coherency in more detail:

- *Address mapping changes*

- *Instruction cache coherency* on page B5-11

- *Direct Memory Access (DMA) operations* on page B5-12

- *Other memory coherency issues* on page B5-13.

5.5.1 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 B3 *Memory Management Unit*), there are two implementation choices for the address associated with a cache line:

- It can be the virtual address of the data in the cache line. This is the more usual choice, because it allows cache line look-up to proceed in parallel with address translation.

- 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 an IMPLEMENTATION DEFINED sequence of cache and/or write buffer operations before a change of virtual-to-physical address mapping. Typically, this sequence contains 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.

There might also be requirements for the code that performs the change of address mapping and any data it accesses to be uncachable, unbufferable or both.

5.5.2 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 needs to be reloaded.

This problem can be avoided by performing an IMPLEMENTATION DEFINED 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 memory, this IMPLEMENTATION DEFINED sequence must be executed before the instruction is executed. A typical case where this needs to 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, the IMPLEMENTATION DEFINED 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 needs to 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 *Instruction Memory Barriers (IMBs)* on page A2-28 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. Whether this is possible is IMPLEMENTATION DEFINED.

- 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.

5.5.3 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:

- marking the memory areas involved in the DMA operation as uncachable 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.

5.5.4 Other memory coherency issues

Memory coherency issues not covered above are those involving the data cache, main memory and/or the write buffer, and which do not involve a change of virtual-to-physical address mapping or a DMA operation. All such issues must be dealt with automatically by the memory system, so that the value returned to the ARM processor is the most up-to-date of the values in the possible physical locations.

——— **Note** ———

This requirement applies to a single processor only. If a system contains multiple ARM processors, all issues relating to memory coherency between the separate processors are system-dependent.

5.6   CP15 registers

Caches and write buffers are controlled with System Control coprocessor registers 7 and 9, and some bits of register 1.

5.6.1   Cache and write buffer control bits in register 1

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 and ignores writes. If the cache cannot be disabled, this bit reads as 1 and ignores writes.

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 reads as 0 and ignore writes. If the write buffer cannot be disabled, this bit reads as 1 and ignores writes.

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 reads as 0 and ignores writes. If the instruction cache cannot be disabled, this bit reads as 1 and ignores writes.

RR (bit[14])   If the cache allows an alternative replacement strategy to be used which 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).

         ARM DDI 0100E

5.6.2   Register 7: Cache functions

The System Control coprocessor register 7 is a write-only register which is used to control 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 5-2 lists the functions available by using an `MCR p15,0,<Rd>,c7,<CRm>,<opcode2>` instruction to write to register 7.

Writing to register 7 with a combination of `<CRm>` and `<opcode2>` which is not listed in Table 5-2 has UNPREDICTABLE results.

Not all functions listed in Table 5-2 are necessarily relevant to or supported by an implementation. Attempting to perform a function which is not supported by the implementation has UNPREDICTABLE results.

Reading register 7 with an `MRC` instruction has UNPREDICTABLE results.

In Table 5-2, 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.
Invalidate	Means that the cache line (or all the lines in the cache) is marked as invalid, so that no cache hits 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 (which must be cache line aligned) is loaded into the cache.
Drain write buffer	Stops the ARM from executing further until all data in the write buffer has been stored to main memory. It can be used instead of unbufferable memory when the timing of specific main memory stores needs to be controlled (for example, when a store to an interrupt acknowledge location needs to complete before interrupts are enabled).
Wait for interrupt	Puts the ARM into a low power state and stops it executing further until an interrupt occurs.
	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`.
Prefetch buffer	Operations are IMPLEMENTATION DEFINED.
Branch target cache	Operations are IMPLEMENTATION DEFINED.
Data	Is the value that is written to register 7, which is the value in the register `<Rd>` specified in the `MCR` instruction.
	If the data is stated to be a virtual address, it must be cache line aligned. This means that if the cache line length is 2^L bytes, bits[L-1:0] of the address must be zero. This address is looked up in the cache. If a cache hit occurs, the specified operation

---

occurs on the cache line it identifies. If a cache miss occurs, nothing more is done. A loop of operations of this type can be used to clean and/or invalidate all cache lines relating to a specified range of addresses.

— **Note** —

If the *Fast Context Switch Extension* (FCSE) described in Chapter B6 is being used, the virtual addresses passed to CP15 by some of the functions in Table 5-2 on page B5-16 are passed as data. This means that no address modification is performed for them by the FCSE.

If the data is stated to be *set/index*, it identifies the cache line that the operation is to be applied to by specifying which cache set it belongs to and what its index is within the set. A loop of operations of this type can be used to clean and/or invalidate an entire cache, or all of the non-locked down portion of the cache (see *Register 9: Cache lockdown* on page B5-18).

If 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, the format of set/index data is:

31	32-A 31-A	L+S L+S-1	L L-1 0
index	SBZ	set	SBZ
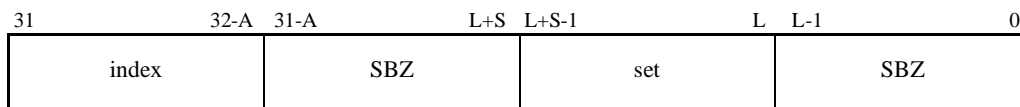
For the definitions of the cache size parameters, see *Cache size* on page B5-4. If the implementation includes a Cache Type register, these parameters can be determined from it (see *Cache Type register* on page B2-9).

Table 5-2 Cache and similar functions

<CRm>	<opcode2>	Function	Data
c0	4	Wait for interrupt	SBZ
c5	0	Invalidate entire instruction cache	SBZ
c5	1	Invalidate instruction cache line	Virtual address
c5	2	Invalidate instruction cache line	Set/index
c5	4	Flush prefetch buffer	SBZ
c5	6	Flush entire branch target cache	SBZ
c5	7	Flush branch target cache entry	IMP
c6	0	Invalidate entire data cache	SBZ
c6	1	Invalidate data cache line	Virtual address

Table 5-2 Cache and similar functions (Continued)

\<CRm\>	\<opcode2\>	Function	Data
c6	2	Invalidate data cache line	Set/index
c7	0	Invalidate entire unified cache or both instruction and data caches	SBZ
c7	1	Invalidate unified cache line	Virtual address
c7	2	Invalidate unified cache line	Set/index
c8	2	Wait for interrupt (alternative, deprecated encoding)	SBZ
c10	1	Clean data cache line	Virtual address
c10	2	Clean data cache line	Set/index
c10	4	Drain write buffer	SBZ
c11	1	Clean unified cache line	Virtual address
c11	2	Clean unified cache line	Set/index
c13	1	Prefetch instruction cache line	Virtual address
c14	1	Clean and invalidate data cache line	Virtual address
c14	2	Clean and invalidate data cache line	Set/index
c15	1	Clean and invalidate unified cache line	Virtual address
c15	2	Clean and invalidate unified cache line	Set/index

5.6.3   Register 9: Cache lockdown

One problem with caches is that while they normally improve average access time to data and instructions, they usually make the worst-case access time worse. 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 due to 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 some ARM memory systems designed to alleviate this. It allows critical code and data (such as 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.

In the description that follows, LINELEN, ASSOCIATIVITY and NSETS are the cache size parameters described in *Cache size* on page B5-4.

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 index as the lockdown block (see *Set-associativity* on page B5-4). So lockdown block 0 consists of the cache line with index 0 from each cache set, etc.

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.

───── **Note** ─────

You cannot lock down the entire cache. At least one lockdown block must be left 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.)

The System Control coprocessor primary register 9 is used to control cache lockdown. If separate caches are used, there are two variants of this register, selected by the `<opcode2>` field of the `MCR` or `MRC` instruction used to access register 9:

`<opcode2> == 0`   Selects the data cache lockdown register.

`<opcode2> == 1`   Selects the instruction cache lockdown register.

If a unified cache is used, there is only one variant of this register and `<opcode2>` should be 0.

`<CRm>` must always be `c0` for `MCR` and `MRC` instructions that access register 9.

Formats for the cache lockdown register

There are two possible formats for the cache lockdown register. These are shown in the following text as Format A and Format B, and it is IMPLEMENTATION DEFINED which one an implementation uses.

Both formats include an index field, which is chosen to be wide enough to hold a lockdown block index. So, its width W is the logarithm base 2 of ASSOCIATIVITY, rounded up to an integer if necessary.

If the implementation includes a Cache Type register, it can be used to determine the register 9 format (see *Cache Type register* on page B2-9).

Format A:

31 32-W	31-W 0
index	UNP/SBZ
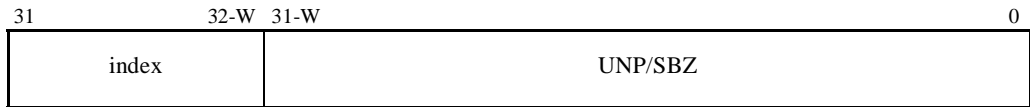
Reading a Format A register returns the value last written to it.

Writing a Format A register has the following effects:

- The next cache miss in each cache set replaces the cache line with the specified index in that cache set.

- The replacement strategy for the cache is constrained so that it can only select cache lines with the specified index and higher, until the register is written again.

Format B:

31	30 W	W-1 0
L	UNP/SBZ	index
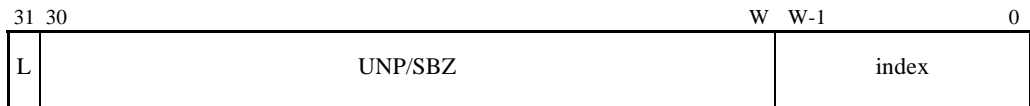
Reading a Format B register returns the value last written to it.

Writing a Format B register has the following effects:

- If L == 1, all cache misses replace the cache line with the specified index 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 index 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 index and higher, until the register is written again.

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, 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 uncachable 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 uncachable area of memory, apart from the data which is to be locked down.

4. Ensure that the data/instructions that are to be locked down are in a cachable 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 index == 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, 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.

7. Write to register 9 with index == N (for Formats A and B), and L == 0 (for Format B).

--- **Note** ---

If the *Fast Context Switch Extension* (FCSE) described in Chapter B6 is being used, care needs to 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 minimise 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.

Cache unlock procedure

To unlock the locked-down portion of the cache, write to register 9 with index == 0 (for Formats A and B), and L == 0 (for Format B).

 ARM DDI 0100E

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 cache contents affect locked-down cache lines as normal. If such operations are used, ensure that they do not use virtual addresses or cache set/index combinations that affect the locked-down cache lines. (Or 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 5-2 on page B5-16) which identify a cache line by set/index can disrupt the constraints on the normal replacement strategy. To avoid this affecting the locked-down area of the cache, read register 9 immediately before code that uses these operations, and write the value read back to register 9 immediately afterwards.

Chapter B6
Fast Context Switch Extension

This chapter describes the *Fast Context Switch Extension* (FCSE). It contains the following sections:

6.1 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 B3 *Memory Management Unit*). 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.

 ARM DDI 0100E

6.2 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 6-1:
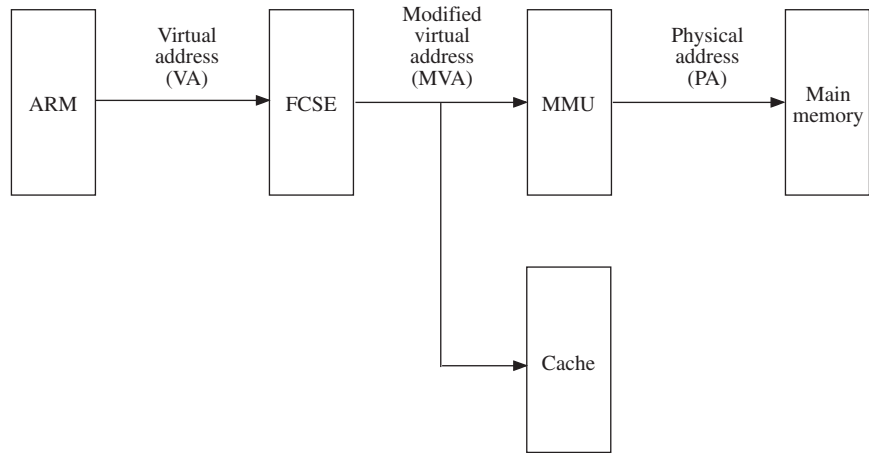
Figure 6-1 Address flow in MMU memory system with FCSE

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 *Register 7: Cache functions* on page B5-15. For these operations, no address modification occurs, and MVA = VA.

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 *Domains* on page B3-17).

         ARM DDI 0100E

6.3 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.

6.4 CP15 registers

The FCSE only uses one coprocessor 15 register, namely register 13.

6.4.1 Register 13: Process ID

```
31              25 24                                                    0
┌─────────────────┬───────────────────────────────────────────────────┐
│      PID        │                   UNP/SBZP                         │
└─────────────────┴───────────────────────────────────────────────────┘
```
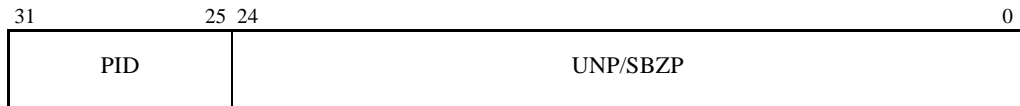
Reading register 13 returns the PID in bits[31:25]. Bits[24:0] of the value read are UNPREDICTABLE.

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 A2-28. 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 A2-30.

────────────────

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 C1-3
- *Compliance with the IEEE 754 standard* on page C1-7
- *IEEE 754 implementation choices* on page C1-8.

1.1 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.

To date, there has only been one major version of the VFP architecture (Version 1, or VFPv1). 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.

1.2   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*.

1.2.1   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 C1-4).

1.2.2   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.

1.2.3   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.

1.2.4   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.

1.2.5 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 will fail to respond to these instructions, causing the ARM's Undefined Instruction exception to occur. For more details, see *Undefined Instruction exception* on page A2-15.

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 A3-5), 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 A2-15), and they will not normally be 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 will deny 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.

1.3   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).

1.4   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.

1.4.1   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.

1.4.2   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.

1.4.3 Comparison results

The results of comparison instructions are delivered as condition codes. In particular, they are flag combinations (N,Z,C,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,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.

1.4.4 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.

1.4.5 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:

2.1 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.

2.1.1 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 $< \mathtt{0xFF}$, the value is a *normalized number* and is equal to:

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

  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 $1 \leq$ 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 C2-18).

2.1.2 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]	
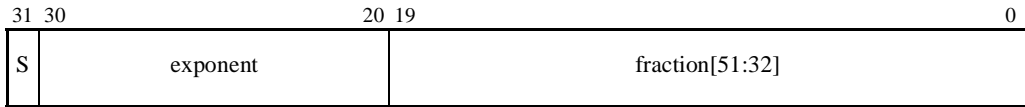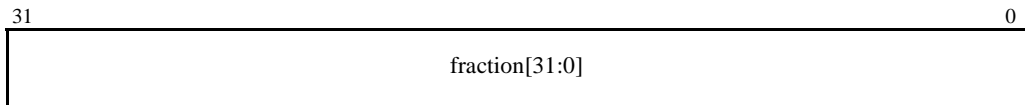
When held in memory, the two words must appear consecutively and must both be word-aligned. The order of the two words depends on the endianness of the memory system:

- 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 *Endianness* on page A2-23 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 $0 < \text{exponent} < 0x7FF$, 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:

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

  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^{-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:

   — 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.

2.1.3 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 *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
```

2.2   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).

2.3   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

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 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 2-1 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 C4-58 and *FMXR* on page C4-70).

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.

Table 2-1 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) RM,RZ: $+$MaxNorm	RN,RM: $-\infty$ (minus infinity) RP,RZ: $-$MaxNorm
Underflow	Normal rounded result	Normal rounded result
Inexact	Normal rounded result	Normal rounded result

The following notes apply to Table 2-1:

- 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.

2.3.1 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.

2.4 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 *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.

2.5 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 illustrated in Figure 2-1.

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
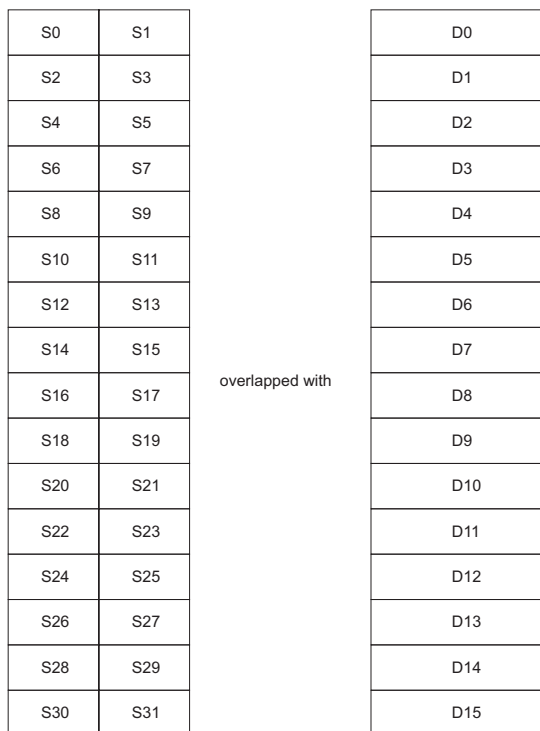
Figure 2-1 VFP general-purpose registers

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.

2.5.1   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 in order 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.

- 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
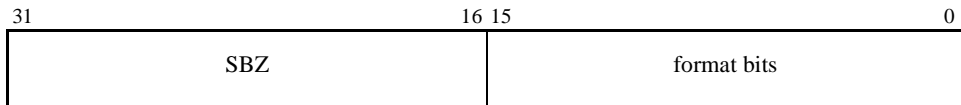
   For each double-precision register Di which is stored by the FSTMX instruction, bit[i] of the format word indicates how it is stored:

   — 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 2-2 on page C2-17 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 2-2 Standard FSTMX/FLDMX memory formats

In the second column of Figure 2-2, 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	

2.5.2 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.

2.5.3 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.

                   ARM DDI 0100E

2.6   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 C2-20 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 C2-21 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 C2-24 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 may not be used outside the implementation itself, except as described in implementation-specific documentation.

2.6.1   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	
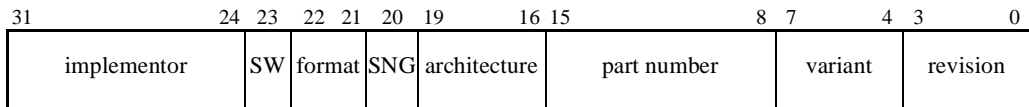
Bits[31:24]   Contain an implementor code. The following code is defined:

> `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 C2-15):

> **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.

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.

2.6.2   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	FZ	RMODE	STRIDE	DNM	LEN	DNM	IXE	UFE	OFE	DZE	IOE	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, in order 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.

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 2-2 on page C2-23.

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 2-2 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 2-2 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

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 2-3 shows which bits are associated with each exception.

Table 2-3 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])

2.6.3   FPEXC

The FPEXC register has the following format:

31 30 29		0
EX EN	IMPLEMENTATION DEFINED	
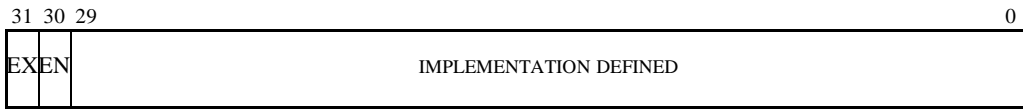
This register can only be accessed in privileged modes.

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.

2.7   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
- *Register transfer instructions* on page C3-17.

3.1   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 3-1 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 3-2 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 3-1 and Table 3-2 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 3-1 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 3-2 on page C3-4	See Table 3-2 on page C3-4	Extension instructions

Table 3-2 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 $\leftrightarrow$ double precision conversions
1000	0	FUITOS	FUITOD	Unsigned integer $\rightarrow$ floating-point conversions
1000	1	FSITOS	FSITOD	Signed integer $\rightarrow$ floating-point conversions
1001	x	-	-	UNDEFINED
101x	x	-	-	UNDEFINED
1100	0	FTOUIS	FTOUID	Floating-point $\rightarrow$ unsigned integer conversions
1100	1	FTOUIZS	FTOUIZD	Floating-point $\rightarrow$ unsigned integer conversions, RZ mode
1101	0	FTOSIS	FTOSID	Floating-point $\rightarrow$ signed integer conversions
1101	1	FTOSIZS	FTOSIZD	Floating-point $\rightarrow$ signed integer conversions, RZ mode
111x	x	-	-	UNDEFINED

3.1.1   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.

3.1.2   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.

3.1.3 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 3-3:

Table 3-3 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 3-4 shows how each predicate must be tested to get the correct results according to the IEEE 754 standard:

Table 3-4 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

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.

3.1.4   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 → 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 → integer conversion is wanted.

All of the floating-point → 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.

3.1.5 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.

3.2 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 A5-56. In addition, a VFP implementation uses them to determine which load/store operation is required, as shown in Table 3-5 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 A5-56.

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 C2-15).

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 A5-56.

The least significant bit of this offset also helps to determine which load/store operation is required, as shown in Table 3-5 on page C3-14. In addition, for Load Multiple and Store Multiple instructions, the offset determines how many registers are to be transferred.

Table 3-5 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 3-5 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 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.

3.2.1 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 A5-56, only the Immediate offset mode (see *Load and Store Coprocessor - Immediate offset* on page A5-58) 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 writeback is available.

3.2.2 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 C2-15). 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 A5-64). 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 A5-62). 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 A5-60). 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.

────────────────

 ARM DDI 0100E

3.3 Register transfer instructions

All VFP 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 3-6.

L bit   This bit determines the direction of the transfer:

- from an ARM register to a VFP register (an `MCR` instruction, with L == 0)
- from a VFP register to an ARM register (an `MRC` instruction, with L == 1).

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 3-7 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 3-6 shows the assignment of register transfer opcodes and other details of the instructions:

Table 3-6 VFP 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 3-7 shows how system registers are encoded in FMXR and FMRX instructions:

Table 3-7 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 0100E

3.3.1 General-purpose 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.

3.3.2 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 C2-19 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).

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.

4.1 Alphabetical list of VFP instructions

Each VFP instruction is described in detail on the following pages.

4.1.1 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 A3-5. 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.

4.1.2 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	
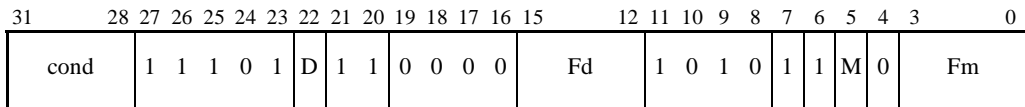
The FABSS (Floating-point Absolute Value, Single-precision) instruction writes the absolute value of a single-precision register to another single-precision register. It can also perform a vector version of this operation.

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 A3-5. 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.

4.1.3 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	
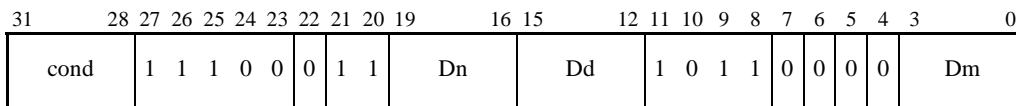
The FADDD (Floating-point Addition, Double-precision) instruction adds together two double-precision registers and writes the result to a third double-precision register. It can also perform a vector version of this operation.

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 A3-5. 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.

4.1.4   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
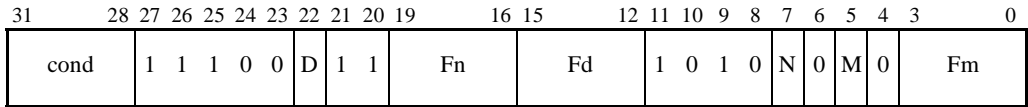
The FADDS (Floating-point Addition, Single-precision) instruction adds together two single-precision registers and writes the result to a third single-precision register. It can also perform a vector version of this operation.

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 A3-5. 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.

4.1.5   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	
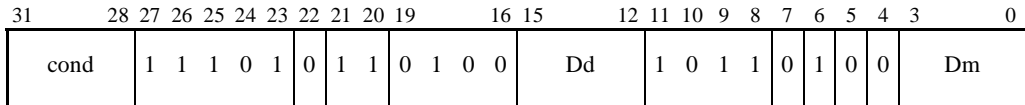
The FCMPD (Floating-point Compare, Double-precision) instruction compares two double-precision registers, writing the result to the FPSCR flags (which is normally transferred to the ARM flags by a subsequent FMSTAT instruction).

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 A3-5. 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.

4.1.6 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		
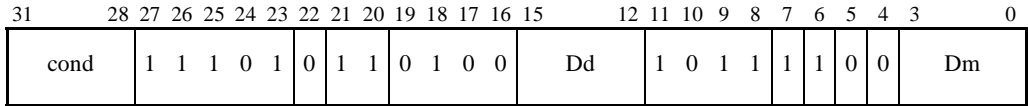
The FCMPED (Floating-point Compare (NaN Exceptions), Double-precision) instruction compares two double-precision registers, writing the result to the FPSCR flags (which is normally transferred to the ARM flags by a subsequent FMSTAT instruction).

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 A3-5. 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.

4.1.7 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	
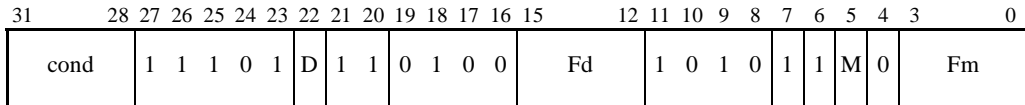
The FCMPES (Floating-point Compare (NaN Exceptions), Single-precision) instruction compares two single-precision registers, writing the result to the FPSCR flags (which is normally transferred to the ARM flags by a subsequent FMSTAT instruction).

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 A3-5. 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.

4.1.8 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 A3-5. 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.

4.1.9　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			
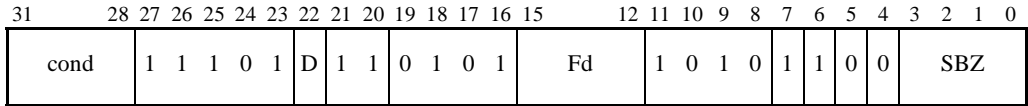
The FCMPEZS (Floating-point Compare (NaN Exceptions) with Zero, Single-precision) instruction compares a single-precision register with zero, writing the result to the FPSCR flags (which is normally transferred to the ARM flags by a subsequent FMSTAT instruction).

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 A3-5. 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.

4.1.10   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			
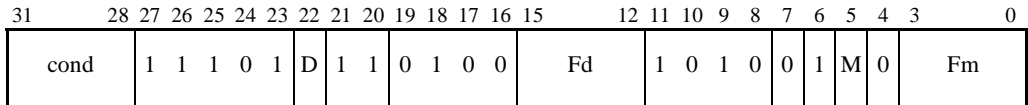
The FCMPS (Floating-point Compare, Single-precision) instruction compares two single-precision registers, writing the result to the FPSCR flags (which is normally transferred to the ARM flags by a subsequent FMSTAT instruction).

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 A3-5. 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.

4.1.11   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 A3-5. 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.

4.1.12   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 A3-5. 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.

4.1.13 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	
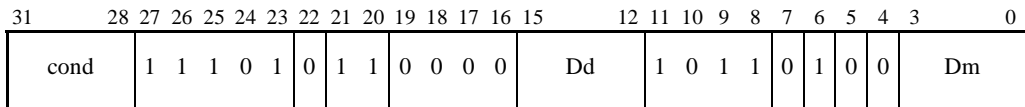
The FCPYD (Floating-point Copy, Double-precision) instruction copies one double-precision register to another double-precision register. It can also perform a vector version of this operation.

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 A3-5. 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 `FCYPD` 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.

4.1.14 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		
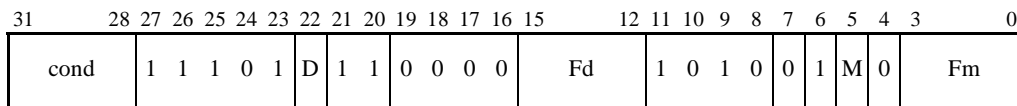
The FCPYS (Floating-point Copy, Single-precision) instruction copies one single-precision register to another single-precision register. It can also perform a vector version of this operation.

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 A3-5. 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 `FCYPS` 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.

4.1.15   **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	
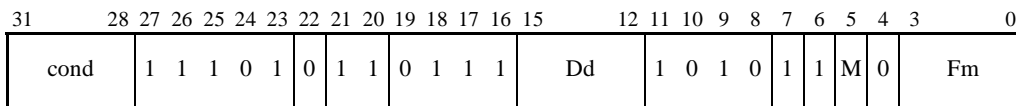
The FCVTDS (Floating-point Convert to Double-precision from Single-precision) instruction converts the
value in a single-precision register to double precision and writes the result to a double-precision register.

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 A3-5. 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.

4.1.16   **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 A3-5. 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.

4.1.17 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		
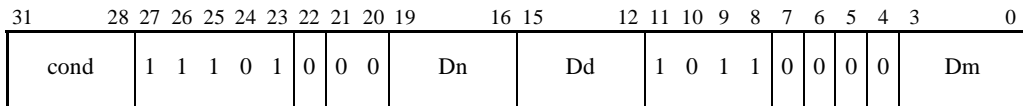
The FDIVD (Floating-point Divide, Double-precision) instruction divides one double-precision register by another double-precision register and writes the result to a third double-precision register. It can also perform a vector version of this operation.

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 A3-5. 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 tradeoff.

Also see *Interrupts* on page C1-5 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.

4.1.18 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
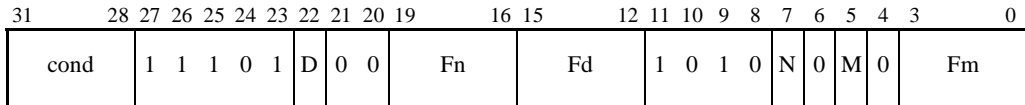
The FDIVS (Floating-point Divide, Single-precision) instruction divides one single-precision register by another single-precision register and writes the result to a third single-precision register. It can also perform a vector version of this operation.

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 A3-5. 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]
```

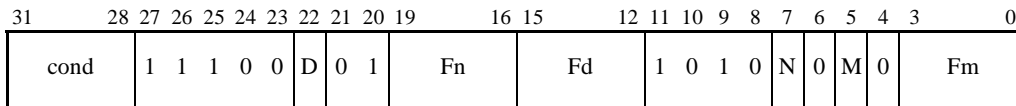## Usage

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 tradeoff.

Also see *Interrupts* on page C1-5 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.

4.1.19   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	
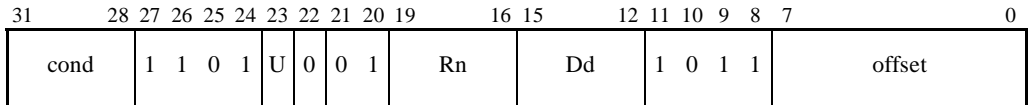
The FLDD (Floating-point Load, Double-precision) instruction loads a double-precision register from memory.

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 A3-5. 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) in order 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.

4.1.20   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	
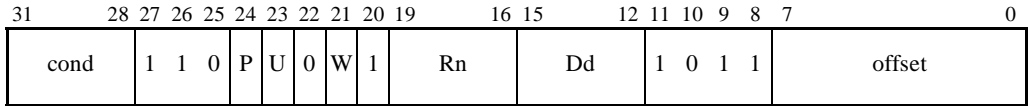
The FLDMD (Floating-point Load Multiple, Double-precision) instruction loads a sequence of consecutive double-precision registers from memory.

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 A3-5. 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.

4.1.21   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	
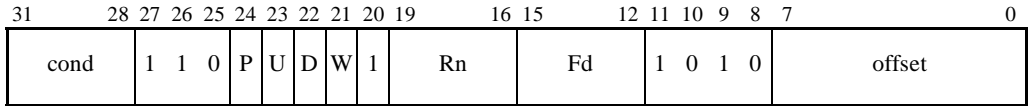
The FLDMS (Floating-point Load Multiple, Single-precision) instruction loads a sequence of consecutive single-precision registers from memory.

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 A3-5. 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).

4.1.22   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	
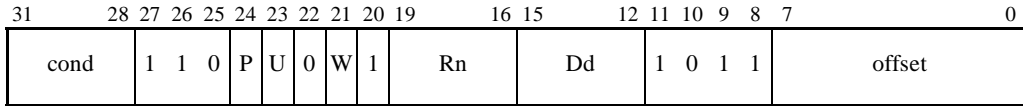
The FLDMX (Floating-point Load Multiple, Unknown precision) instruction loads a sequence of consecutive double-precision registers from memory, using the same IMPLEMENTATION DEFINED format as the FSTMX instruction. This allows the registers to be reloaded correctly with integers, single-precision values 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 A3-5. 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.

4.1.23 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	
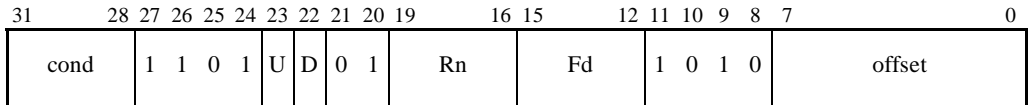
The FLDS (Floating-point Load, Single-precision) instruction loads a single-precision register from memory.

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 A3-5. 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) in order 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 needs to 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).

4.1.24   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		
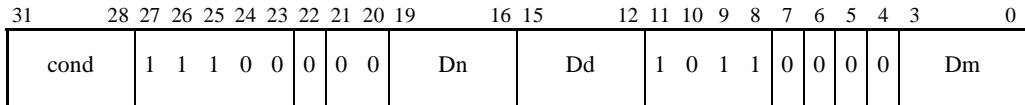
The FMACD (Floating-point Multiply and Accumulate, Double-precision) instruction multiplies together two double-precision registers, adds a third double-precision register to the product and writes the result to the third register. It can also perform a vector version of this operation.

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 A3-5. 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]
```
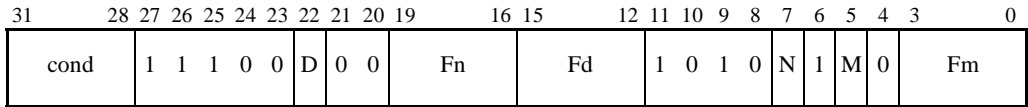
Notes

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.

4.1.25 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		
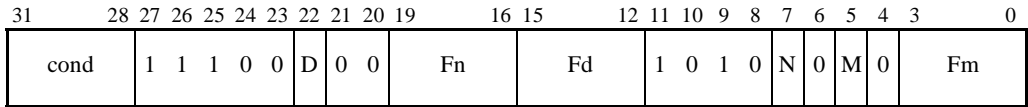
The FMACS (Floating-point Multiply and Accumulate, Single-precision) instruction multiplies together two single-precision registers, adds a third single-precision register to the product and writes the result to the third register. It can also perform a vector version of this operation.

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 A3-5. 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]
```
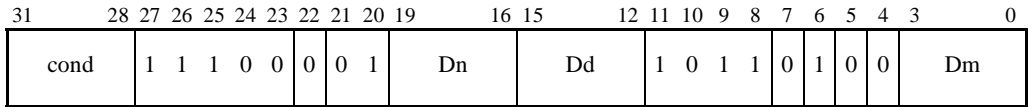
Notes

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.

4.1.26 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 A3-5. 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.

4.1.27 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	
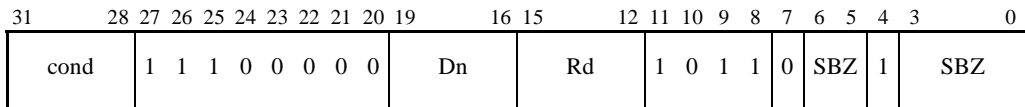
The FMDLR (Floating-point Move to Double-precision Low from Register) instruction transfers the contents of the ARM register Rd to the lower half of the double-precision register Dn. Used with FMDHR, it transfers double-precision values between ARM registers and floating-point registers.

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 A3-5. 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.

4.1.28 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 A3-5. 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.

4.1.29 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 A3-5. 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.

4.1.30　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	
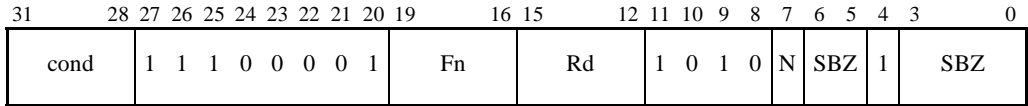
The FMRS (Floating-point Move to Register from Single-precision) instruction transfers the contents of the single-precision register Fn to the ARM register Rd. The value transferred can be an integer (typically generated by a FTOSID, FTOSIS, FTOUID or FTOUIS instruction) or a single-precision floating-point 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 A3-5. 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
```

　　　　Copyright © 1996-2000 ARM Limited. All rights reserved.　　　　ARM DDI 0100E

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 needs to 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.

4.1.31   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 A3-5. 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 trap.

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 trap 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.

4.1.32   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 A3-5. 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.

4.1.33   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 A3-5. 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.

4.1.34 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	
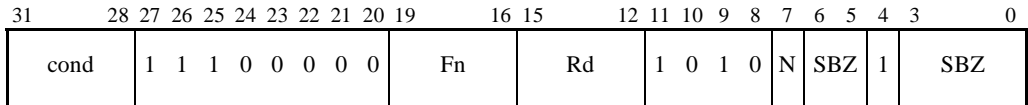
The FMSR (Floating-point Move to Single-precision from Register) instruction transfers the contents of the ARM register Rd to the single-precision register Fn. The value transferred can subsequently be treated either as an integer (if used as the source register of a FSITOD, FSITOS, FUITOD or FUITOS 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 A3-5. 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).

4.1.35 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 A3-5. 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 C4-58.

4.1.36 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	
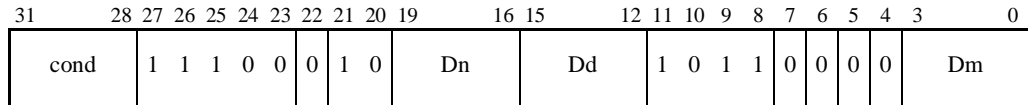
The FMULD (Floating-point Multiply, Double-precision) instruction multiplies together two double-precision registers and writes the result to a third double-precision register. It can also perform a vector version of this operation.

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 A3-5. 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.

4.1.37 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	
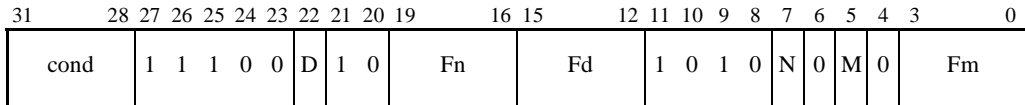
The FMULS (Floating-point Multiply, Single-precision) instruction multiplies together two single-precision registers and writes the result to a third single-precision register. It can also perform a vector version of this operation.

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 A3-5. 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.

4.1.38 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 A3-5. 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
```

Copyright © 1996-2000 ARM Limited. All rights reserved.

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 trap.

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 trap 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.

4.1.39   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	
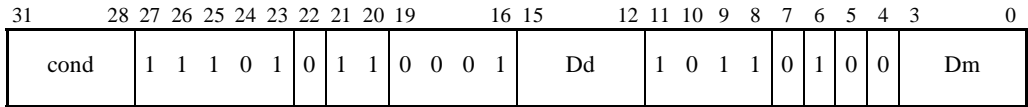
The FNEGD (Floating-point Negate, Double-precision) instruction negates the value of a double-precision register and writes the result to another double-precision register. It can also perform a vector version of this operation.

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 A3-5. 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.

4.1.40 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	
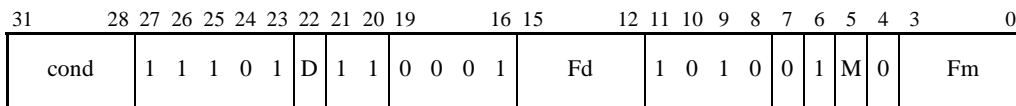
The FNEGS (Floating-point Negate, Single-precision) instruction negates the value of a single-precision register and writes the result to another single-precision register. It can also perform a vector version of this operation.

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 A3-5. 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 FNEDS 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.

4.1.41 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	
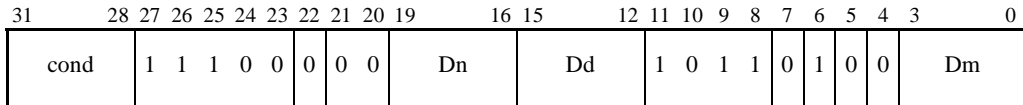
The FNMACD (Floating-point Negated Multiply and Accumulate, Double-precision) instruction multiplies together two double-precision registers, adds a third double-precision register to the negation of the product and writes the result to the third register. It can also perform a vector version of this 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 A3-5. 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
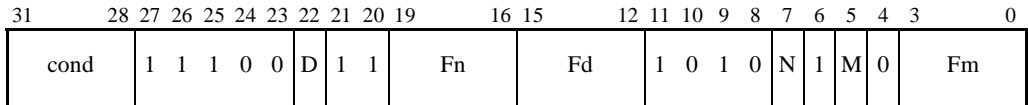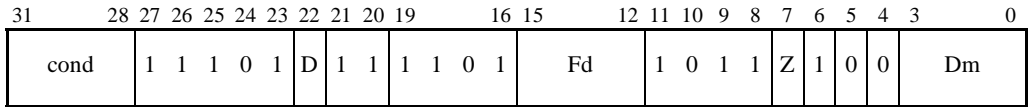
Operation

```
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.

4.1.42 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 A3-5. 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.

4.1.43 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 A3-5. 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.

4.1.44 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 A3-5. 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.

4.1.45   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	
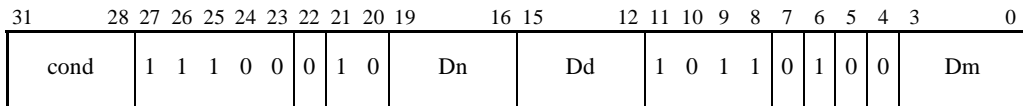
The FNMULD (Floating-point Negated Multiply, Double-precision) instruction multiplies together two double-precision registers, and writes the negation of the result to a third double-precision register. It can also perform a vector version of this operation.

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 A3-5. 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.

4.1.46 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	
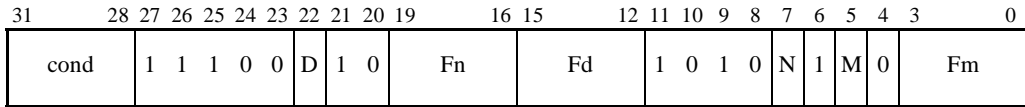
The FNMULS (Floating-point Negated Multiply, Single-precision) instruction multiplies together two single-precision registers, and writes the negation of the result to a third single-precision register. It can also perform a vector version of this operation.

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 A3-5. 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.

4.1.47 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	
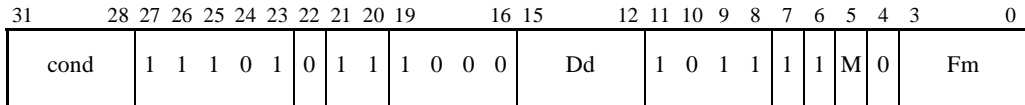
The FSITOD (Floating-point Convert Signed Integer to Double-precision) instruction converts a signed integer value held in a single-precision register to double precision and writes the result to a double-precision register. The integer value will normally have been transferred from memory by a single-precision 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 A3-5. 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.

4.1.48   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 A3-5. 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.

4.1.49 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	
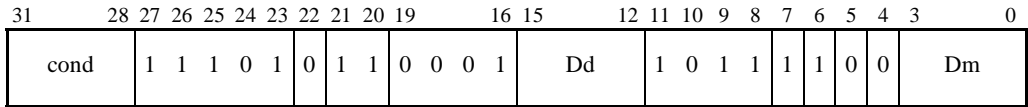
The FSQRTD (Floating-point Square Root, Double-precision) instruction calculates the square root of the value in a double-precision register and writes the result to another double-precision register. It can also perform a vector version of this operation.

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 A3-5. 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 C1-5 for a description of some implications for interrupt latency.

 ARM DDI 0100E

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.

4.1.50 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	
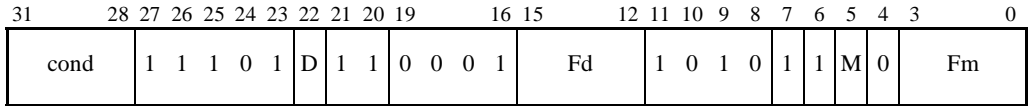
The FSQRTS (Floating-point Square Root, Single-precision) instruction calculates the square root of the value in a single-precision register and writes the result to another single-precision register. It can also perform a vector version of this operation.

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 A3-5. 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 C1-5 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.

4.1.51 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					
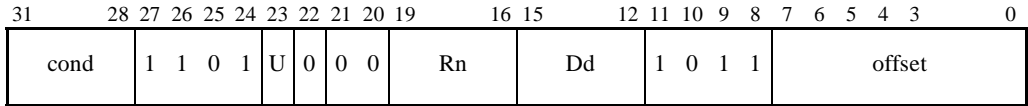
The FSTD (Floating-point Store, Double-precision) instruction stores a double-precision register to memory.

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 A3-5. 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.

4.1.52 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					
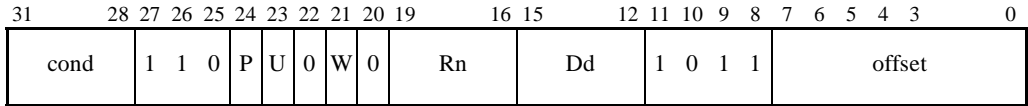
The FSTMD (Floating-point Store Multiple, Double-precision) instruction stores a sequence of consecutive double-precision registers to memory.

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 A3-5. 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.

4.1.53　FSTMS

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	D	W	0	Rn			Fd			1	0	1	0	offset						
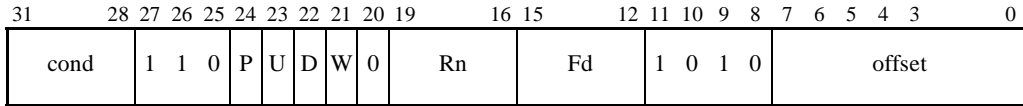
The FSTMS (Floating-point Store Multiple, Single-precision) instruction stores a sequence of consecutive single-precision registers to memory.

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 A3-5. 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 will be 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.

4.1.54   FSTMX

31	28	27	26	25	24	23	22	21	20	19	16	15	12	11	10	9	8	7	3	0
cond		1	1	0	P	U	0	W	0	Rn		Dd		1	0	1	1	offset		
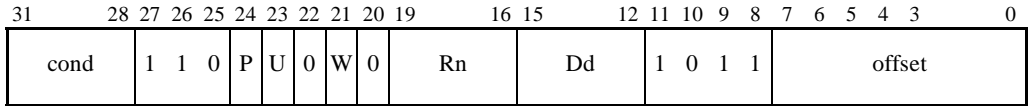
The FSTMX (Floating-point Store Multiple, Unknown precision) instruction stores a sequence of consecutive double-precision registers to memory, in an IMPLEMENTATION DEFINED format that allows the registers to be reloaded correctly regardless of whether they contain integers, single-precision values or 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 A3-5. 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.

4.1.55   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					
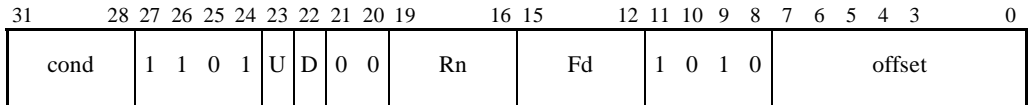
The FSTS (Floating-point Store, Single-precision) instruction stores a single-precision register to memory.

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 A3-5. 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) in order 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.

4.1.56   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	
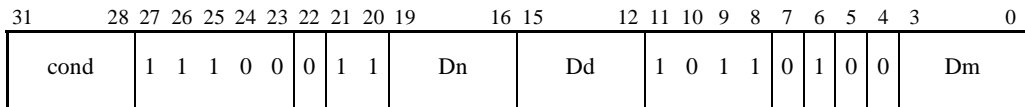
The FSUBD (Floating-point Subtract, Double-precision) instruction subtracts one double-precision register from another double-precision register and writes the result to a third double-precision register. It can also perform a vector version of this operation.

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 A3-5. 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.

4.1.57   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 A3-5. 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.

4.1.58   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 A3-5. 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.

4.1.59   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			
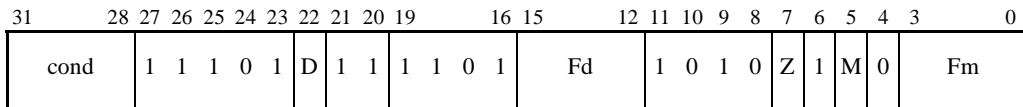
The FTOSIS (Floating-point Convert to Signed Integer from Single-precision) instruction converts a value held in a single-precision register to a signed integer and writes the result to a second single-precision register. The integer value is normally then transferred to memory by a single-precision 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 A3-5. 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 −∞ (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 +∞ (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.

4.1.60 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	
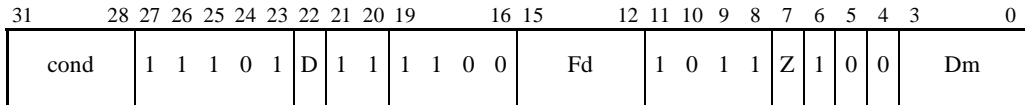
The `FTOUID` (Floating-point Convert to Unsigned Integer from Double-precision) instruction converts a value held in a double-precision register to an unsigned integer and writes the result to a single-precision register. The integer value is normally then transferred to memory by a single-precision 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 A3-5. 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 −∞ (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.

4.1.61 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	
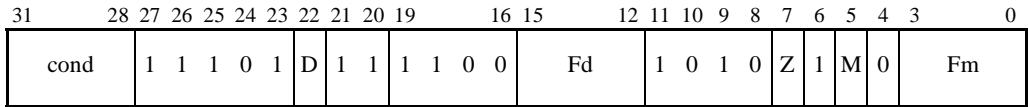
The `FTOUIS` (Floating-point Convert to Unsigned Integer from Single-precision) instruction converts a value held in a single-precision register to an unsigned integer and writes the result to a second single-precision register. The integer value is normally then transferred to memory by a single-precision 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 A3-5. 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.

4.1.62 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		
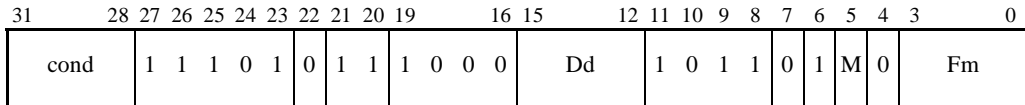
The FUITOD (Floating-point Convert Unsigned Integer to Double-precision) instruction converts an unsigned integer value held in a single-precision register to double precision and writes the result to a double-precision register. The integer value will normally have been transferred from memory by a 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 A3-5. 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.

4.1.63 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 A3-5. 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:

- *Addressing Mode 1 - Single-precision vectors (non-monadic)* on page C5-2
- *Addressing Mode 2 - Double-precision vectors (non-monadic)* on page C5-8
- *Addressing Mode 3 - Single-precision vectors (monadic)* on page C5-14
- *Addressing Mode 4 - Double-precision vectors (monadic)* on page C5-19
- *Addressing Mode 5 - VFP load/store multiple* on page C5-24.

5.1 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	
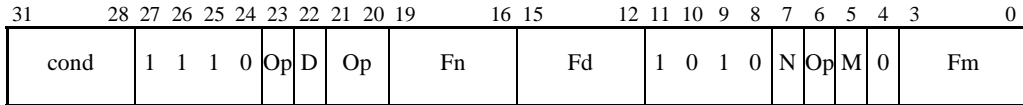
When the vector length indicated by the FPSCR is greater than 1, the single-precision two-operand instructions FADDS, FDIVS, FMULS, FNMULS,and FSUBS can specify three different types of behavior:

- 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
```

5.1.1 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 C2-22). 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 5-1.
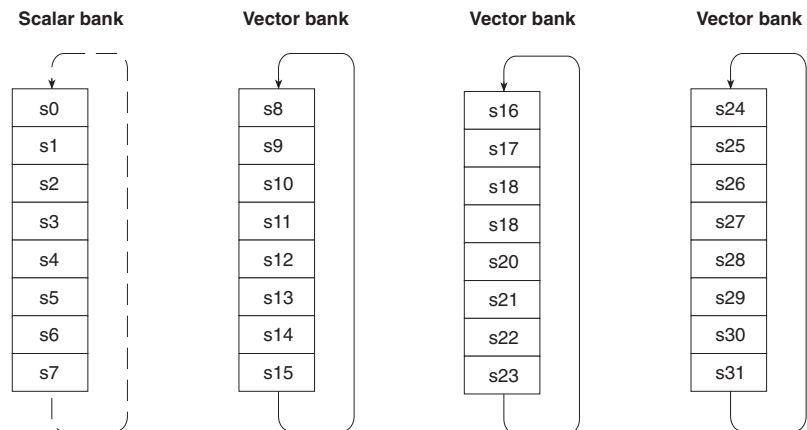
Figure 5-1 Single-precision register banks

5.1.2   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
```

           ARM DDI 0100E

5.1.3   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.

5.1.4   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.

5.1.5 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.

5.2 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
```

5.2.1 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 C2-22). 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 5-2.
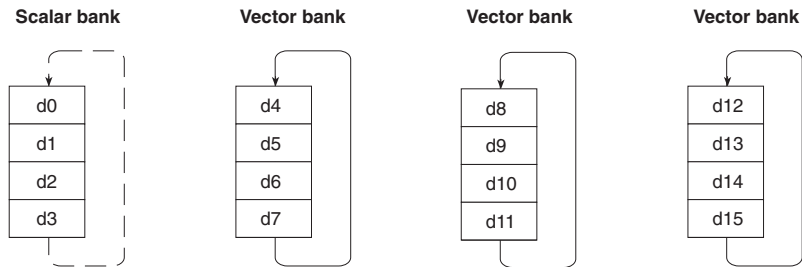
Figure 5-2 Double-precision register banks

5.2.2   Operation

The following pages describe each of the three possible forms of the addressing mode:

*   *Scalar operations* on page C5-11
*   *Mixed vector/scalar operations* on page C5-12
*   *Vector operations* on page C5-13.

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
```

 ARM DDI 0100E

5.2.3   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.

5.2.4   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.

                    ARM DDI 0100E

5.2.5 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.

5.3   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.

5.3.1 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
```

5.3.2   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.

5.3.3   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.

5.3.4 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.

5.4 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.

Copyright © 1996-2000 ARM Limited. All rights reserved.

5.4.1 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
```

5.4.2 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.

5.4.3 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.

5.4.4 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.

5.5 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
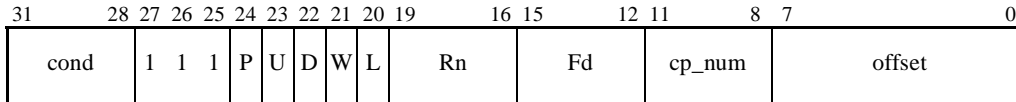
The VFP load multiple instructions (FLDMD, FLDMS, FLDMX) are examples of ARM LDC instructions, whose addressing modes are described in *Addressing Mode 5 - Load and Store Coprocessor* on page A5-56. Similarly, the VFP store multiple instructions (FSTMD, FSTMS, FSTMX) are examples of ARM STC instructions, 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.

5.5.1 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 5-1 shows details of this.

Table 5-1 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 5-1 mean the coprocessor does not respond to the instruction, which make the ARM's Undefined Instruction exception occur (see *Undefined Instruction exception* on page A2-15).

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.

5.5.2   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 C4-425 and *FSTMX* on page C4-100). 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 A3-5. 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.

5.5.3 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 C4-42 and *FSTMX* on page C4-100). 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 A3-5), 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 A3-5. 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 writeback 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.

5.5.4   VFP load/store multiple - Decrement
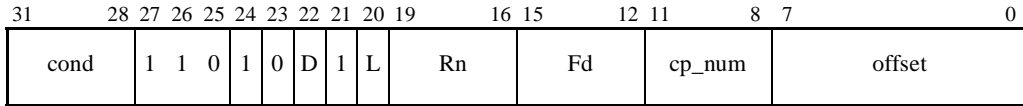
31	28	27	26	25	24	23	22	21	20	19	16	15	12	11	8	7	0
cond		1	1	0	1	0	D	1	L	Rn		Fd		cp_num		offset	

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 C4-42 and *FSTMX* on page C4-100). 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 A3-5.

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 A3-5. 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 writeback that occurs in this addressing mode. It cannot be omitted, as no non-writeback 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.

    ARM DDI 0100E

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.

5.5.5 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 A5-54 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 5-2 shows the relationship between the non-stacking and stacking names of the instructions:

Table 5-2 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 A5-56).

It is therefore recommended that these stack types are not used on systems that use the VFP architecture.

Glossary

26-bit architecture

Means earlier versions of the ARM architecture which implement only a 26-bit address space. These are versions ARMv1, ARMv2, and ARMv2a.

32-bit architecture

Means versions of the ARM architecture which implement a 32-bit address space. These are ARM architecture version 3 and above.

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 writeback. For more details, see *Effects of data-aborted instructions* on page A2-17.

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.

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 writeback
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.

Boolean AND
Is signified by the AND operator.

Boolean OR
Is signified by the OR operator.

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.

                   ARM DDI 0100E

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.

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.

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.

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.

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.

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.

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 the PSR as well in the 26-bit architectures), 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.

Interworking
> Is a method of working that allows branches between ARM and Thumb code.

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.

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.

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 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.

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.

MMU   *See* Memory Management Unit.

MVA   *See* Modified Virtual Address.

NaN   Means Not a Number, and is a type of floating-point value.

                   ARM DDI 0100E

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 (or bits[25:2] of R15 on 26-bit architectures).

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 (or bits[31:26] and bits[1:0] of register 15 on 26-bit architectures).

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.

                   ARM DDI 0100E

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.

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 $-2^{(n-1)} \le x \le 2^{(n-1)} - 1$

- $2^{(n-1)} - 1$ 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.

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.

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 appropriately word-aligned or halfword-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 trap. See *Undefined Instruction exception* on page A2-15 for information on undefined instruction traps.

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.

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.

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.

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.

  ARM DDI 0100E

 ARM DDI 0100E

   ARM DDI 0100E

Numerics