ARM® v7-M Architecture Reference Manual

ARM®

ARM DDI 0403C_errata_v3 (ID021910)

ARMv7-M Architecture Reference Manual

Release Information

The following changes have been made to this document.

Change history

Date	Issue	Confidentiality	Change
June 2006	A	Non-confidential	Initial release
July 2007	B	Non-confidential	Second release, errata and changes documented separately
September 2008	C	Non-confidential, Restricted Access	Options for additional watchpoint based trace in the DWT, plus errata updates and clarifications.
July 2009	C_errata	Non-confidential	Marked-up errata PDF, see page iii for more information.
February 2010	C_errata_v3	Non-confidential	Additional marked-up errata PDF, see page iii for more information.

Proprietary Notice

This ARM Architecture Reference Manual is protected by copyright and the practice or implementation of the information herein may be protected by one or more patents or pending applications. No part of this ARM Architecture Reference Manual may be reproduced in any form by any means without the express prior written permission of ARM. **No license, express or implied, by estoppel or otherwise to any intellectual property rights is granted by this ARM Architecture Reference Manual**.

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

This ARM Architecture Reference Manual is provided “as is”. ARM makes no representations or warranties, either express or implied, included but not limited to, warranties of merchantability, fitness for a particular purpose, or non-infringement, that the content of this ARM Architecture Reference Manual is suitable for any particular purpose or that any practice or implementation of the contents of the ARM Architecture Reference Manual will not infringe any third party patents, copyrights, trade secrets, or other rights.

This ARM Architecture Reference Manual may include technical inaccuracies or typographical errors.

To the extent not prohibited by law, in no event will ARM be liable for any damages, including without limitation any direct loss, lost revenue, lost profits or data, special, indirect, consequential, incidental or punitive damages, however caused and regardless of the theory of liability, arising out of or related to any furnishing, practicing, modifying or any use of this ARM Architecture Reference Manual, even if ARM has been advised of the possibility of such damages.

Words and logos marked with ® or ™ are registered trademarks or trademarks of ARM Limited, except as otherwise stated below in this proprietary notice. Other brands and names mentioned herein may be the trademarks of their respective owners.

Copyright © 2006-2010 ARM Limited

110 Fulbourn Road Cambridge, England CB1 9NJ

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

Note

The term ARM is also used to refer to versions of the ARM architecture, for example ARMv6 refers to version 6 of the ARM architecture. The context makes it clear when the term is used in this way.

Note

- This errata PDF is regenerated from the source files of issue C of this document, but:
  - Some pseudocode examples, that are imported into the document, have been updated. Markups highlight significant changes in these pseudocode inserts.

    Other pseudocode updates are made using the standard Acrobat editing tools.
  - Pages ii and iii of the PDF have been replaced, by an edit to the PDF, to include an updated **Proprietary Notice**.

  With these exceptions, this PDF corresponds to the released PDF of issue C of the document, with errata indicated by markups to the PDF:
  - the original errata markups, issued June 2009, are identified as ARM_2009_Q2
  - additional errata markups, issued February 2010, are identified as ARM_2009_Q4.
- In the revised pseudocode, the function `BadReg(x)` is replaced by a new construct, `x IN {13,15}`, that can be used in other contexts. This is a format change only.
- From February 2010, issue C of the ARMv7-M ARM is superseded by issue D of the document. ARM strongly recommends you to use issue D of the document in preference to using this errata PDF.

Contents
ARMv7-M Architecture Reference Manual

Preface

Part A Application Level Architecture

Chapter A1 Introduction

Chapter A2 Application Level Programmers' Model

Chapter A3 ARM Architecture Memory Model

List of Tables
ARMv7-M Architecture Reference Manual

List of Figures
ARMv7-M Architecture Reference Manual

Preface

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

About this manual

This manual documents the Microcontroller profile associated with version 7 of the ARM® Architecture (ARMv7-M). For short-form definitions of all the ARMv7 profiles see page A1-1.

The manual consists of three parts:

Part A The application level programming model and memory model information along with the instruction set as visible to the application programmer.

This is the information required to program applications or to develop the toolchain components (compiler, linker, assembler and disassembler) excluding the debugger. For ARMv7-M, this is almost entirely a subset of material common to the other two profiles. Instruction set details which differ between profiles are clearly stated.

Note

All ARMv7 profiles support a common procedure calling standard, the ARM Architecture Procedure Calling Standard (AAPCS).

Part B The system level programming model and system level support instructions required for system correctness. The system level supports the ARMv7-M exception model. It also provides features for configuration and control of processor resources and management of memory access rights.

This is the information in addition to Part A required for an operating system (OS) and/or system support software. It includes details of register banking, the exception model, memory protection (management of access rights) and cache support.

Part B is profile specific. ARMv7-M introduces a new programmers' model and as such has some fundamental differences at the system level from the other profiles. As ARMv7-M is a memory-mapped architecture, the system memory map is documented here.

Part C The debug features to support the ARMv7-M debug architecture and the programmer's interface to the debug environment.

This is the information required in addition to Parts A and B to write a debugger. Part C covers details of the different types of debug:

- halting debug and the related Debug state
- exception-based monitor debug
- non-invasive support for event generation and signalling of the events to an external agent.

This part is profile specific and includes several debug features unique within the ARMv7 architecture to this profile.

Using this manual

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

Part A, Application level architecture

Part A describes the application level view of the architecture. It contains the following chapters:

Chapter A1 ***Introduction***

ARMv7 overview, the different architecture profiles and the background to the Microcontroller (M) profile.

Chapter A2 ***Application Level Programmers' Model***

Details on the registers and status bits available at the application level along with a summary of the exception support.

Chapter A3 ***ARM Architecture Memory Model***

Details of the ARM architecture memory attributes and memory order model.

Chapter A4 ***The ARMv7-M Instruction Set***

General information on the Thumb® instruction set.

Chapter A5 ***Thumb Instruction Set Encoding***

Encoding diagrams for the Thumb instruction set along with information on bit field usage, UNDEFINED and UNPREDICTABLE terminology.

Chapter A6 ***Thumb Instruction Details***

Contains detailed reference material on each Thumb instruction, arranged alphabetically by instruction mnemonic. Summary information for system instructions is included and referenced for detailed definition in Part B.

Part B, system level architecture

Part B describes the system level view of the architecture. It contains the following chapters:

Chapter B1 ***System Level Programmers' Model***

Details of the registers, status and control mechanisms available at the system level.

Chapter B2 ***System Memory Model***

Details of the pseudocode used to support memory accesses to the ARM architecture memory model.

Chapter B3 ***System Address Map***

Overview of the system address map and details of the architecturally defined features within the Private Peripheral Bus region. This chapter includes details of the memory-mapped support for a protected memory system.

Chapter B4 ***ARMv7-M System Instructions***

Contains detailed reference material on the system level instructions.

Part C, debug architecture

Part C describes the debug architecture. It contains the following chapter:

Chapter C1 ***ARMv7-M Debug***

ARMv7-M debug support.

Part D, appendices

This manual contains a glossary and the following appendices:

Appendix A ***CPUID***

The revised format for ARM architecture CPUID registers including the description and associated values of all attribute fields relevant to the ARMv7-M architecture. Attribute values are used to describe instruction set and memory model support of an architecture variant. Some attribute values reflect architectural choice for an implementation.

Appendix B ***ARMv7-M infrastructure IDs***

A summary of the ARM CoreSight™ compatible ID registers used for ARM architecture infrastructure identification.

Appendix C ***Legacy Instruction Mnemonics***

A cross reference of Unified Assembler Language forms of the instruction syntax to the Thumb format used in earlier versions of the ARM architecture.

Appendix D ***Deprecated Features in ARMv7-M***

Deprecated features that software is advised to avoid for future proofing. ARM intends to remove this functionality in a future version of the ARM architecture.

Appendix E ***Debug ITM and DWT packet protocol***

The debug trace packet protocol used to export ITM and DWT sourced information.

Appendix F ***ARMv7-R differences***

A summary of differences between the ARMv7-R and ARMv7-M profiles.

Appendix G ***Pseudocode definition***

Definition of terms, format and helper functions used by the pseudocode to describe the memory model and instruction operations.

Appendix H ***Pseudocode Index***

Index to definitions of pseudocode operators, keywords, functions, and procedures.

Appendix I *Register Index*

Index to register descriptions in the manual

Glossary

Glossary of terms - does not include terms associated with pseudocode.

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, pseudocode descriptions of instructions, and source code examples. For more details of the conventions used in assembler syntax descriptions see *Assembler syntax* on page A6-4. For more details of pseudocode conventions see Appendix G *Pseudocode definition*.

The `typewriter` font is also used in the main text for instruction mnemonics and for references to other items appearing in assembler syntax descriptions, pseudocode 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.

Further reading

This section lists publications that provide additional information on the ARM architecture and ARM family of processors. This manual provides architecture information, the contract between hardware and software for development of ARM compliant cores, compiler and debug tools development and software to run on the ARM targets. The Technical Reference Manual (TRM) for the implementation of interest provides details of the IMPLEMENTATION DEFINED architecture features in the *ARM compliant core*. The silicon partner's device specification should be used for additional system details.

ARM periodically provides updates and corrections to its documentation. For the latest information and errata, some materials are published at `http://www.arm.com`. Alternatively, contact your distributor, or silicon partner who will have access to the latest published ARM information, as well as information specific to the device of interest. Your local ARM office has access to the latest published ARM information.

ARM publications

This document is specific to the ARMv7-M architecture. Other relevant publications relating to ARMv7-M implementations and ARM's debug architecture are:

- *Cortex-M3 Technical Reference Manual* (ARM DDI 0337)
- *Procedure Call Standard for the ARM Architecture* (ARM GENC 003534)
- *ARM Debug Interface v5 Architecture Specification* (ARM IHI 0031)
- *CoreSight Architecture Specification* (ARM IHI 0029)
- *Embedded Trace Macrocell Architecture Specification* (ARM DDI 0014).

For information on ARMv6-M, see the *ARMv6-M Architecture Reference Manual* (ARM DDI 0419).

For information on the ARMv7-A and -R profiles, see the *ARM Architecture Reference Manual* (ARM DDI 0406).

Feedback

ARM welcomes feedback on its documentation.

Feedback on this book

If you notice any errors or omissions in this book, send email to `errata@arm.com` giving:

- the document title
- the document number
- the page number(s) to which your comments apply
- a concise explanation of the problem.

General suggestions for additions and improvements are also welcome.

Part A

Application Level Architecture

Chapter A1
Introduction

ARMv7 is documented as a set of architecture profiles. Three profiles have been defined as follows:

ARMv7-A the application profile for systems supporting the ARM and Thumb instruction sets, and requiring virtual address support in the memory management model.

ARMv7-R the realtime profile for systems supporting the ARM and Thumb instruction sets, and requiring physical address only support in the memory management model

ARMv7-M the microcontroller profile for systems supporting only the Thumb instruction set, and where overall size and deterministic operation for an implementation are more important than absolute performance.

While profiles were formally introduced with the ARMv7 development, the A-profile and R-profile have implicitly existed in earlier versions, associated with the Virtual Memory System Architecture (VMSA) and Protected Memory System Architecture (PMSA) respectively.

Instruction Set Architecture (ISA)

ARMv7-M only supports execution of Thumb instructions. For a detailed list of the instructions supported, see Chapter A6 *Thumb Instruction Details*.

A1.1 The ARM Architecture – M profile

The ARM architecture has evolved through several major revisions to a point where it supports implementations across a wide spectrum of performance points, with over a billion parts per annum being produced. The latest version (ARMv7) has seen the diversity formally recognized in a set of architecture profiles, the profiles used to tailor the architecture to different market requirements. A key factor is that the application level is consistent across all profiles, and the bulk of the variation is at the system level.

The introduction of Thumb-2 technology in ARMv6T2 provided a balance to the ARM and Thumb instruction sets, and the opportunity for the ARM architecture to be extended into new markets, in particular the microcontroller marketplace. To take maximum advantage of this opportunity a Thumb-only profile with a new programmers' model (a system level consideration) has been introduced as a unique profile, complementing ARM's strengths in the high performance and real-time embedded markets.

Key criteria for ARMv7-M implementations are as follows:

- Enable implementations with industry leading power, performance and area constraints
  - Opportunities for simple pipeline designs offering leading edge system performance levels in a broad range of markets and applications
- Highly deterministic operation
  - Single/low cycle execution
  - Minimal interrupt latency (short pipelines)
  - Cacheless operation
- Excellent C/C++ target – aligns with ARM's programming standards in this area
  - Exception handlers are standard C/C++ functions, entered using standard calling conventions
- Designed for deeply embedded systems
  - Low pincount devices
  - Enable new entry level opportunities for the ARM architecture
- Debug and software profiling support for event driven systems

This manual is specific to the ARMv7-M profile.

Chapter A2
Application Level Programmers' Model

This chapter provides an application level view of the programmers' model. This is the information necessary for application development, as distinct from the system information required to service and support application execution under an operating system. It contains the following sections:

System related information is provided in overview form and/or with references to the system information part of the architecture specification as appropriate.

A2.1 About the Application level programmers' model

This chapter contains the programmers' model information required for application development.

The information in this chapter is distinct from the system information required to service and support application execution under an operating system. That information is given in Chapter B1 *System Level Programmers' Model*.

A2.1.1 Privileged execution

System level support requires access to all features and facilities of the architecture, a level of access generally referred to as privileged operation. System code determines whether an application runs in a privileged or unprivileged manner. When an operating system supports both privileged and unprivileged operation, an application usually runs unprivileged. This:

- permits the operating system to allocate system resources to it in a unique or shared manner
- provides a degree of protection from other processes and tasks, and so helps protect the operating system from malfunctioning applications.

A2.1.2 System level architecture

Thread mode is the fundamental mode for application execution in ARMv7-M and is selected on reset. Thread mode can raise a supervisor call using the SVC instruction or handle system access and control directly.

All exceptions execute in Handler mode. Supervisor call (SVCall) handlers manage resources on behalf of the application such as interaction with peripherals, memory allocation and management of software stacks.

This chapter only provides a limited amount of system level information. Where appropriate it:

- gives an overview of the system level information
- gives references to the system level descriptions in Chapter B1 *System Level Programmers' Model* and elsewhere.

A2.2 ARM core data types and arithmetic

ARMv7-M processors support the following data types in memory:

Byte	8 bits
Halfword	16 bits
Word	32 bits

Processor registers are 32 bits in size. The instruction set contains instructions supporting the following data types held in registers:

- 32-bit pointers
- unsigned or signed 32-bit integers
- unsigned 16-bit or 8-bit integers, held in zero-extended form
- signed 16-bit or 8-bit integers, held in sign-extended form
- unsigned or signed 64-bit integers held in two registers.

Load and store operations can transfer bytes, halfwords, or words to and from memory. Loads of bytes or halfwords zero-extend or sign-extend the data as it is loaded, as specified in the appropriate load instruction.

The instruction sets include load and store operations that transfer two or more words to and from memory. You can load and store 64-bit integers using these instructions.

When any of the data 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.

Direct instruction support for 64-bit integers is limited, and most 64-bit operations require sequences of two or more instructions to synthesize them.

A2.2.1 Integer arithmetic

The instruction set provides a wide variety of operations on the values in registers, including bitwise logical operations, shifts, additions, subtractions, multiplications, and many others. These operations are defined using the *pseudocode* described in Appendix G *Pseudocode definition*, usually in one of three ways:

- By direct use of the pseudocode operators and built-in functions defined in *Operators and built-in functions* on page AppxG-11.
- By use of pseudocode helper functions defined in the main text.
- By a sequence of the form:
  1. Use of the `SInt()`, `UInt()`, and `Int()` built-in functions defined in *Converting bitstrings to integers* on page AppxG-14 to convert the bitstring contents of the instruction operands to the unbounded integers that they represent as two's complement or unsigned integers.
  2. Use of mathematical operators, built-in functions and helper functions on those unbounded integers to calculate other such integers.
  3. Use of either the bitstring extraction operator defined in *Bitstring extraction* on page AppxG-12 or of the saturation helper functions described in *Pseudocode details of saturation* on page A2-9 to convert an unbounded integer result into a bitstring result that can be written to a register.

Shift and rotate operations

The following types of shift and rotate operations are used in instructions:

Logical Shift Left

(LSL) moves each bit of a bitstring left by a specified number of bits. Zeros are shifted in at the right end of the bitstring. Bits that are shifted off the left end of the bitstring are discarded, except that the last such bit can be produced as a carry output.

Logical Shift Right

(LSR) moves each bit of a bitstring right by a specified number of bits. Zeros are shifted in at the left end of the bitstring. Bits that are shifted off the right end of the bitstring are discarded, except that the last such bit can be produced as a carry output.

Arithmetic Shift Right

(ASR) moves each bit of a bitstring right by a specified number of bits. Copies of the leftmost bit are shifted in at the left end of the bitstring. Bits that are shifted off the right end of the bitstring are discarded, except that the last such bit can be produced as a carry output.

Rotate Right (ROR) moves each bit of a bitstring right by a specified number of bits. Each bit that is shifted off the right end of the bitstring is re-introduced at the left end. The last bit shifted off the the right end of the bitstring can be produced as a carry output.

Rotate Right with Extend

(RRX) moves each bit of a bitstring right by one bit. The carry input is shifted in at the left end of the bitstring. The bit shifted off the right end of the bitstring can be produced as a carry output.

Pseudocode details of shift and rotate operations

These shift and rotate operations are supported in pseudocode by the following functions:

```
// LSL_C()
// =======

(bits(N), bit) LSL_C(bits(N) x, integer shift)
    assert shift > 0;
    extended_x = x : Zeros(shift);
    result = extended_x<N-1:0>;
    carry_out = extended_x<N>;
    return (result, carry_out);

// LSL()
// =====

bits(N) LSL(bits(N) x, integer shift)
    assert shift >= 0;
    if shift == 0 then
        result = x;
    else
```

```
        (result, -) = LSL_C(x, shift);
    return result;

// LSR_C()
// =======

(bits(N), bit) LSR_C(bits(N) x, integer shift)
    assert shift > 0;
    extended_x = ZeroExtend(x, shift+N);
    result = extended_x<shift+N-1:shift>;
    carry_out = extended_x<shift-1>;
    return (result, carry_out);

// LSR()
// =====

bits(N) LSR(bits(N) x, integer shift)
    assert shift >= 0;
    if shift == 0 then
        result = x;
    else
        (result, -) = LSR_C(x, shift);
    return result;

// ASR_C()
// =======

(bits(N), bit) ASR_C(bits(N) x, integer shift)
    assert shift > 0;
    extended_x = SignExtend(x, shift+N);
    result = extended_x<shift+N-1:shift>;
    carry_out = extended_x<shift-1>;
    return (result, carry_out);

// ASR()
// =====

bits(N) ASR(bits(N) x, integer shift)
    assert shift >= 0;
    if shift == 0 then
        result = x;
    else
        (result, -) = ASR_C(x, shift);
    return result;

// ROR_C()
// =======

(bits(N), bit) ROR_C(bits(N) x, integer shift)
    assert shift != 0;
    m = shift MOD N;
    result = LSR(x,m) OR LSL(x,N-m);
    carry_out = result<N-1>;
    return (result, carry_out);
```

```
// ROR()
// =====

bits(N) ROR(bits(N) x, integer shift)
    if n == 0 then
        result = x;
    else
        (result, -) = ROR_C(x, shift);
    return result;

// RRX_C()
// =======

(bits(N), bit) RRX_C(bits(N) x, bit carry_in)
    result = carry_in : x<N-1:1>;
    carry_out = x<0>;
    return (result, carry_out);

// RRX()
// =====

bits(N) RRX(bits(N) x, bit carry_in)
    (result, -) = RRX_C(x, carry_in);
    return result;
```

Pseudocode details of addition and subtraction

In pseudocode, addition and subtraction can be performed on any combination of unbounded integers and bitstrings, provided that if they are performed on two bitstrings, the bitstrings must be identical in length. The result is another unbounded integer if both operands are unbounded integers, and a bitstring of the same length as the bitstring operand(s) otherwise. For the precise definition of these operations, see *Addition and subtraction* on page AppxG-15.

The main addition and subtraction instructions can produce status information about both unsigned carry and signed overflow conditions. This status information can be used to synthesize multi-word additions and subtractions. In pseudocode the `AddWithCarry()` function provides an addition with a carry input and carry and overflow outputs:

```
// AddWithCarry()
// ==============

(bits(N), bit, bit) AddWithCarry(bits(N) x, bits(N) y, bit carry_in)
    unsigned_sum = UInt(x) + UInt(y) + UInt(carry_in);
    signed_sum   = SInt(x) + SInt(y) + UInt(carry_in);
    result       = unsigned_sum<N-1:0>;  // == signed_sum<N-1:0>
    carry_out    = if UInt(result) == unsigned_sum then '0' else '1';
    overflow     = if SInt(result) == signed_sum then '0' else '1';
    return (result, carry_out, overflow);
```

An important property of the `AddWithCarry()` function is that if:

```
(result, carry_out, overflow) = AddWithCarry(x, NOT(y), carry_in)
```

then:

- If `carry_in == '1'`, then `result == x-y` with `overflow == '1'` if signed overflow occurred during the subtraction and `carry_out == '1'` if unsigned borrow did not occur during the subtraction (that is, if `x >= y`).
- If `carry_in == '0'`, then `result == x-y-1` with `overflow == '1'` if signed overflow occurred during the subtraction and `carry_out == '1'` if unsigned borrow did not occur during the subtraction (that is, if `x > y`).

Together, these mean that the `carry_in` and `carry_out` bits in `AddWithCarry()` calls can act as *NOT borrow* flags for subtractions as well as *carry* flags for additions.

Pseudocode details of saturation

Some instructions perform *saturating arithmetic*, that is, if the result of the arithmetic overflows the destination signed or unsigned N-bit integer range, the result produced is the largest or smallest value in that range, rather than wrapping around modulo 2^N. This is supported in pseudocode by the `SignedSatQ()` and `UnsignedSatQ()` functions when a boolean result is wanted saying whether saturation occurred, and by the `SignedSat()` and `UnsignedSat()` functions when only the saturated result is wanted:

```
// SignedSatQ()
// ============

(bits(N), boolean) SignedSatQ(integer i, integer N)
    if i > 2^(N-1) - 1 then
        result = 2^(N-1) - 1;  saturated = TRUE;
    elsif i < -(2^(N-1)) then
        result = -(2^(N-1));  saturated = TRUE;
    else
        result = i;  saturated = FALSE;
    return (result<N-1:0>, saturated);

// UnsignedSatQ()
// ==============

(bits(N), boolean) UnsignedSatQ(integer i, integer N)
    if i > 2^N - 1 then
        result = 2^N - 1;  saturated = TRUE;
    elsif i < 0 then
        result = 0;  saturated = TRUE;
    else
        result = i;  saturated = FALSE;
    return (result<N-1:0>, saturated);

// SignedSat()
// ===========

bits(N) SignedSat(integer i, integer N)
    (result, -) = SignedSatQ(i, N);
    return result;

// UnsignedSat()
// =============

bits(N) UnsignedSat(integer i, integer N)
    (result, -) = UnsignedSatQ(i, N);
    return result;
```

`SatQ(i, N, unsigned)` returns either `UnsignedSatQ(i, N)` or `SignedSatQ(i, N)` depending on the value of its third argument, and `Sat(i, N, unsigned)` returns either `UnsignedSat(i, N)` or `SignedSat(i, N)` depending on the value of its third argument:

```
// SatQ()
// ======

(bits(N), boolean) SatQ(integer i, integer N, boolean unsigned)
    (result, sat) = if unsigned then UnsignedSatQ(i, N) else SignedSatQ(i, N);
    return (result, sat);
// Sat()
// =====

bits(N) Sat(integer i, integer N, boolean unsigned)
    result = if unsigned then UnsignedSat(i, N) else SignedSat(i, N);
    return result;
```

A2.3 Registers and execution state

The application level programmers' model provides details of the general-purpose and special-purpose registers visible to the application programmer, the ARM memory model, and the instruction set used to load registers from memory, store registers to memory, or manipulate data (data operations) within the registers.

Applications often interact with external events. A summary of the types of events recognized in the architecture, along with the mechanisms provided in the architecture to interact with events, is included in *Exceptions, faults and interrupts* on page A2-15). How events are handled is a system level topic described in *Exception model* on page B1-14.

A2.3.1 ARM core registers

There are thirteen general-purpose 32-bit registers (R0-R12), and an additional three 32-bit registers which have special names and usage models.

SP stack pointer (R13), used as a pointer to the active stack. For usage restrictions see *Use of 0b1101 as a register specifier* on page A5-4. This is preset to the top of the Main stack on reset. See *The SP registers* on page B1-8 for additional information.

LR link register (R14), used to store a value (the Return Link) relating to the return address from a subroutine which is entered using a Branch with Link instruction. This register is set to an illegal value (all 1's) on reset. The reset value will cause a fault condition to occur if a subroutine return call is attempted from it. The LR register is also updated on exception entry, see *Exception entry behavior* on page B1-21.

Note

R14 can be used for other purposes when the register is not required to support a return from a subroutine.

PC program counter. For details on the usage model of the PC see *Use of 0b1111 as a register specifier* on page A5-3. The PC is loaded with the Reset handler start address on reset.

Pseudocode details of ARM core register operations

In pseudocode, the `R[]` function is used to:

- Read or write R0-R12, SP, and LR, using n == 0-12, 13, and 14 respectively.
- Read the PC, using n == 15.

This function has prototypes:

```
bits(32) R[integer n]
    assert n >= 0 && n <= 15;

R[integer n] = bits(32) value
    assert n >= 0 && n <= 14;
```

For more details on the R[] function, see *Pseudocode details for ARM core register access in the Thumb instruction set* on page B1-12. Writing an address to the PC causes either a simple branch to that address or an *interworking* branch that, in ARMv7-M, must select the Thumb instruction set to execute after the branch.

Note

The following pseudocode defines behavior in ARMv7-M. It is much simpler than the equivalent pseudo-function definitions that apply to older ARM architecture variants and other profiles.

A simple branch is performed by the `BranchWritePC()` function:

```
// BranchWritePC()
// ===============

BranchWritePC(bits(32) address)
    BranchTo(address<31:1>:’0’);
```

An interworking branch is performed by the `BXWritePC()` function:

```
// BXWritePC()
// ===========

BXWritePC(bits(32) address)
    if CurrentMode == Mode_Handler && address<31:28> == '1111' then
        ExceptionReturn(address<27:0>);
    else
        EPSR.T = address<0>;  // if EPSR.T == 0, a UsageFault('Invalid State')
                              // is taken on the next instruction
        BranchTo(address<31:1>:'0');
```

The `LoadWritePC()` and `ALUWritePC()` functions are used for two cases where the behavior was systematically modified between architecture versions. The functions simplify to aliases of the branch functions in the M-profile architecture variants:

```
// LoadWritePC()
// =============

LoadWritePC(bits(32) address)
    BXWritePC(address);


// ALUWritePC()
// ============

ALUWritePC(bits(32) address)
    BranchWritePC(address);
```

A2.3.2 The Application Program Status Register (APSR)

Program status is reported in the 32-bit Application Program Status Register (APSR), where the defined bits break down into a set of flags as follows:

31	30	29	28	27	26	0
N	Z	C	V	Q	RESERVED	

APSR bit fields are in the following two categories:

- Reserved bits are allocated to system features or are available for future expansion. Further information on currently allocated reserved bits is available in *The special-purpose program status registers (xPSR)* on page B1-8. Application level software must ignore values read from reserved bits, and preserve their value on a write. The bits are defined as UNK/SBZP.
- Flags that can be set by many instructions:

  N, bit [31] Negative condition code flag. Set to bit [31] of the result of the instruction. If the 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, bit [30] Zero condition code flag. Set to 1 if the result of the instruction is zero, and to 0 otherwise. A result of zero often indicates an equal result from a comparison.

  C, bit [29] Carry condition code flag. Set to 1 if the instruction results in a carry condition, for example an unsigned overflow on an addition.

  V, bit [28] Overflow condition code flag. Set to 1 if the instruction results in an overflow condition, for example a signed overflow on an addition.

  Q, bit [27] Set to 1 if an `SSAT` or `USAT` instruction changes (saturates) the input value for the signed or unsigned range of the result.

A2.3.3 Execution state support

ARMv7-M only executes Thumb instructions, and therefore always executes instructions in Thumb state. See Chapter A6 *Thumb Instruction Details* for a list of the instructions supported.

In addition to normal program execution, there is a Debug state – see Chapter C1 *ARMv7-M Debug* for more details.

A2.3.4 Privileged execution

Good system design practice requires the application developer to have a degree of knowledge of the underlying system architecture and the services it offers. System support requires a level of access generally referred to as privileged operation. The system support code determines whether applications run in a privileged or unprivileged manner. Where both privileged and unprivileged support is provided by an operating system, applications usually run unprivileged, allowing the operating system to allocate system resources for sole or shared use by the application, and to provide a degree of protection with respect to other processes and tasks.

Thread mode is the fundamental mode for application execution in ARMv7-M. Thread mode is selected on reset, and can execute in a privileged or unprivileged manner depending on the system environment. Privileged execution is required to manage system resources in many cases. When code is executing unprivileged, Thread mode can execute an `SVC` instruction to generate a supervisor call exception. Privileged execution in Thread mode can raise a supervisor call using `SVC` or handle system access and control directly.

All exceptions execute as privileged code in Handler mode. See *Exception model* on page B1-14 for details. Supervisor call handlers manage resources on behalf of the application such as interaction with peripherals, memory allocation and management of software stacks.

A2.4 Exceptions, faults and interrupts

An exception can be caused by the execution of an exception generating instruction or triggered as a response to a system behavior such as an interrupt, memory management protection violation, alignment or bus fault, or a debug event. Synchronous and asynchronous exceptions can occur within the architecture.

How events are handled is a system level topic described in *Exception model* on page B1-14.

A2.4.1 System related events

The following types of exception are system related. Where there is direct correlation with an instruction, reference to the associated instruction is made.

Supervisor calls are used by application code to request a service from the underlying operating system. Using the SVC instruction, the application can instigate a supervisor call for a service requiring privileged access to the system.

Several forms of Fault can occur:

- Instruction execution related errors
- Data memory access errors can occur on any load or store
- Usage faults from a variety of execution state related errors. Execution of an UNDEFINED instruction is an example cause of a UsageFault exception.
- Debug events can generate a DebugMonitor exception.

Faults in general are synchronous with respect to the associated executing instruction. Some system errors can cause an imprecise exception where it is reported at a time bearing no fixed relationship to the instruction which caused it.

Interrupts are always treated as asynchronous events with respect to the program flow. System timer (SysTick), a Pending[1] service call (PendSV[2]), and a controller for external interrupts (NVIC) are all defined. See *System timer - SysTick* on page B3-24 for information on the SysTick interrupt, and *Nested Vectored Interrupt Controller (NVIC)* on page B3-28 for information on the interrupt controller. PendSV is supported by the Interrupt Control State register (see *Interrupt Control State Register (ICSR)* on page B3-12).

A BKPT instruction generates a debug event – see *Debug event behavior* on page C1-14 for more information.

For power or performance reasons it can be desirable to either notify the system that an action is complete, or provide a hint to the system that it can suspend operation of the current task. Instruction support is provided for the following:

- Send Event and Wait for Event instructions. See *SEV* on page A6-212 and *WFE* on page A6-276.
- Wait For Interrupt. See *WFI* on page A6-277.

[1] For the definition of a Pending exception, see *Exceptions* on page B1-5.

[2] A service (system) call is used by an application which requires a service from an underlying operating system. The service call associated with PendSV executes when the interrupt is taken. For a service call which executes synchronously with respect to program execution use the SVC instruction.

A2.5 Coprocessor support

An ARMv7-M implementation can optionally support coprocessors. If it does not support them, it treats all coprocessors as non-existent. Coprocessors 8 to 15 (CP8 to CP15) are reserved by ARM. Coprocessors 0 to 7 (CP0 to CP7) are IMPLEMENTATION DEFINED, subject to the coprocessor instruction constraints of the instruction set architecture.

Where a coprocessor instruction is issued to a non-existent or disabled coprocessor, a NOCP UsageFault is generated (see *Fault behavior* on page B1-39).

Unknown instructions issued to an enabled coprocessor generate an UNDEFINSTR UsageFault.

Chapter A3
ARM Architecture Memory Model

This chapter covers the general principles which apply to the ARM memory model. The chapter contains the following sections:

ARMv7-M is a memory-mapped architecture. The address map specific details that apply to ARMv7-M are described in *The system address map* on page B3-2.

A3.1 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. The address space can also be considered as consisting of 2^{31} 16-bit halfwords, each of whose addresses is halfword-aligned, which means that the address is divisible by 2. The halfword whose halfword-aligned address is A consists of the two bytes with addresses A and A+1.

While instruction fetches are always halfword-aligned, some load and store instructions support unaligned addresses. This affects the access address A, such that A[1:0] in the case of a word access and A[0] in the case of a halfword access can have non-zero values.

Address calculations are normally performed using ordinary integer instructions. This means that they normally wrap around if they overflow or underflow the address space. Another way of describing this is that any address calculation is reduced modulo 2^{32}.

Normal sequential execution of instructions effectively calculates:

```
(address_of_current_instruction) +(2 or 4)  /*16- and 32-bit instr mix*/
```

after each instruction to determine which instruction to execute next. If this calculation overflows the top of the address space, the result is UNPREDICTABLE. In ARMv7-M this condition cannot occur because the top of memory is defined to always have the eXecute Never (XN) memory attribute associated with it. See *The system address map* on page B3-2 for more details. An access violation will be reported if this scenario occurs.

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.

`LDC`, `LDM`, `LDRD`, `POP`, `PUSH`, `STC`, `STRD`, and `STM` instructions access a sequence of words at increasing memory addresses, effectively incrementing a memory address by 4 for each register load or store. If this calculation overflows the top of the address space, the result is UNPREDICTABLE.

Any unaligned load or store whose calculated address is such that it would access the byte at `0xFFFFFFFF` and the byte at address `0x00000000` as part of the instruction is UNPREDICTABLE.

A3.1.1 Virtual versus physical addressing

Virtual memory is not supported in ARMv7-M. A virtual address (VA) is always equal to a physical address (PA).

A3.2 Alignment support

The system architecture can choose one of two policies for alignment checking in ARMv7-M:

- Support the unaligned access
- Generate a fault when an unaligned access occurs.

The policy varies with the type of access. An implementation can be configured to force alignment faults for all unaligned accesses (see below).

Writes to the PC are restricted according to the rules outlined in *Use of 0b1111 as a register specifier* on page A5-3.

A3.2.1 Alignment behavior

Address alignment affects data accesses and updates to the PC.

Alignment and data access

The following data accesses always generate an alignment fault:

- Non halfword-aligned `LDREXH` and `STREXH`
- Non word-aligned `LDREX` and `STREX`
- Non word-aligned `LDRD`, `LDMIA`, `LDMDB`, `POP`, and `LDC`
- Non word-aligned `STRD`, `STMIA`, `STMDB`, `PUSH`, and `STC`

The following data accesses support unaligned addressing, and only generate alignment faults when the CCR.UNALIGN_TRP bit is set (see *Configuration and Control Register (CCR)* on page B3-16):

- Non halfword-aligned `LDR{S}H{T}` and `STRH{T}`
- Non halfword-aligned `TBH`
- Non word-aligned `LDR{T}` and `STR{T}`

Note

`LDREXD` and `STREXD` are not supported in ARMv7-M.

Accesses to Strongly Ordered and Device memory types must always be naturally aligned (see *Memory access restrictions* on page A3-26).

The ARMv7-M alignment behavior is described in the following pseudocode:

For register definitions see Appendix I *Register Index*.
For ExceptionTaken() see *Exception entry behavior* on page B1-21.
The other functions are local and descriptive only. For the actual memory access functionality, see MemU[] and MemA[] that are used in the instruction definitions (see Chapter A6 *Thumb Instruction Details*), and defined in *Pseudocode details of general memory system operations* on page B2-3.

```
if IsUnaligned(Address) then          // the data access is to an unaligned address
    if AlignedAccessInstr() then      // the instruction does not support unaligned accesses
        UFSR.UNALIGNED = ‘1’;
        ExceptionTaken(UsageFault);
    else
        if CCR.UNALIGN_TRP then       // trap on all unaligned accesses
            UFSR.UNALIGNED = ‘1’;
            ExceptionTaken(UsageFault);
        else
            UnalignedAccess(Address); // perform an unaligned access
else
    AlignedAccess(Address);           // perform an aligned access
```

Alignment and updates to the PC

All instruction fetches must be halfword-aligned. Any exception return irregularities are captured as an INVSTATE or INVPC UsageFault by the exception return mechanism. See *Fault behavior* on page B1-39.

For exception entry and return:

- exception entry using a vector with bit [0] clear causes an INVSTATE UsageFault
- a reserved EXC_RETURN value causes an INVPC UsageFault
- loading an unaligned value from the stack into the PC on an exception return is UNPREDICTABLE.

For all other cases where the PC is updated:

- bit [0] of the value is ignored when loading the PC[1] using an ADD or MOV instruction
- a BLX, BX, LDR to the PC, POP or LDM including the PC instruction will cause an INVSTATE UsageFault if bit [0] of the value loaded is zero
- loading the PC with a value from a memory location whose address is not word aligned is UNPREDICTABLE.

[1] 16-bit form of the ADD (register) and MOV (register) instructions only, otherwise loading the PC is UNPREDICTABLE.

A3.3 Endian support

The address space rules (*Address space* on page A3-2) require that for an 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 fully specify the mappings between words, halfwords and bytes. A memory system uses one of the following mapping schemes. This choice is known as the endianness of the memory system.

In a *little-endian* memory system the mapping between bytes from memory and the interpreted value in an ARM register is illustrated in Table A3-1.

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

Table A3-1 Little-endian byte format

	31 24	**23 16**	**15 8**	**7 0**
Word at Address A	Byte {Addr + 3}	Byte {Addr + 2}	Byte {Addr + 1}	Byte {Addr + 0}
Halfword at Address A			Byte {Addr + 1}	Byte {Addr + 0}

In a *big-endian* memory system the mapping between bytes from memory and the interpreted value in an ARM register is illustrated in Table A3-2.

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

Table A3-2 Big-endian byte format

	31 24	**23 16**	**15 8**	**7 0**
Word at Address A	Byte {Addr + 0}	Byte {Addr + 1}	Byte {Addr + 2}	Byte {Addr + 3}
Halfword at Address A			Byte {Addr + 0}	Byte {Addr + 1}

For a word address A, Table A3-3 and Table A3-4 show how the word at address A, the halfwords at address A and A+2, and the bytes at addresses A, A+1, A+2 and A+3 map onto each other for each endianness.

Table A3-3 Little-endian memory system

MSByte	MSByte -1	LSByte + 1	LSByte
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

Table A3-4 Big-endian memory system

MSByte	MSByte -1	LSByte + 1	LSByte
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

The big-endian and little-endian mapping schemes determine the order in which the bytes of a word or half-word are interpreted.

As an example, a load of a word (4 bytes) from address `0x1000` will result in an access of the bytes contained at memory locations `0x1000`, `0x1001`, `0x1002` and `0x1003`, regardless of the mapping scheme used. The mapping scheme determines the significance of those bytes.

A3.3.1 Control of the Endian Mapping in ARMv7-M

ARMv7-M supports a selectable endian model, that is configured to be big endian (BE) or little endian (LE) by a control input on a reset. The endian mapping has the following restrictions:

- The endian setting only applies to data accesses, instruction fetches are always little endian
- Loads and stores to the System Control Space (*System Control Space (SCS)* on page B3-6) are always little endian

Where big endian format instruction support is required, it can be implemented in the bus fabric. See *Endian support* on page AppxF-2 for more details.

Instruction alignment and byte ordering

Thumb instruction execution enforces 16-bit alignment on all instructions. This means that 32-bit instructions are treated as two halfwords, hw1 and hw2, with hw1 at the lower address.

In instruction encoding diagrams, hw1 is shown to the left of hw2. This results in the encoding diagrams reading more naturally. The byte order of a 32-bit Thumb instruction is shown in Figure A3-1.

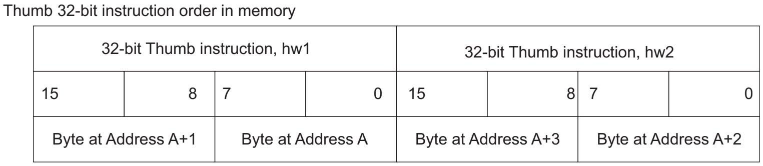

Thumb 32-bit instruction order in memory
32-bit Thumb instruction, hw1
32-bit Thumb instruction, hw2
15
8
7
0
15
8
7
0
Byte at Address A+1
Byte at Address A
Byte at Address A+3
Byte at Address A+2

Figure A3-1 Instruction byte order in memory

A3.3.2 Element size and Endianness

The effect of the endianness mapping on data applies to the size of the element(s) being transferred in the load and store instructions. Table A3-5 shows the element size of each of the load and store instructions:.

Table A3-5 Load-store and element size association

Instruction class	Instructions	Element Size
Load/store byte	`LDR{S}B{T}`, `STRB{T}`, `LDREXB`, `STREXB`	byte
Load/store halfword	`LDR{S}H{T}`, `STRH{T}`, `TBH`, `LDREXH`, `STREXH`	halfword
Load/store word	`LDR{T}`, `STR{T}`, `LDREX`, `STREX`	word
Load/store two words	`LDRD`, `STRD`	word
Load/store multiple words	`LDM{IA,DB}`, `STM{IA,DB}`, `PUSH`, `POP`, `LDC`, `STC`	word

A3.3.3 Instructions to reverse bytes in a general-purpose register

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

ARMv7-M supports instructions for the following byte transformations (see the instruction definitions in Chapter A6 *Thumb Instruction Details* for details):

`REV` Reverse word (four bytes) register, for transforming 32-bit representations.

`REVSH` Reverse halfword and sign extend, for transforming signed 16-bit representations.

`REV16` Reverse packed halfwords in a register for transforming unsigned 16-bit representations.

A3.4 Synchronization and semaphores

Exclusive access instructions support non-blocking shared-memory synchronization primitives that allow calculation to be performed on the semaphore between the read and write phases, and scale for multiprocessor system designs.

In ARMv7-M, the synchronization primitives provided are:

- Load-Exclusives:
  - LDREX, see *LDREX* on page A6-106
  - LDREXB, see *LDREXB* on page A6-107
  - LDREXH, see *LDREXH* on page A6-108
- Store-Exclusives:
  - STREX, see *STREX* on page A6-234
  - STREXB, see *STREXB* on page A6-235
  - STREXH, see *STREXH* on page A6-236
- Clear-Exclusive, CLREX, see *CLREX* on page A6-56.

Note

This section describes the operation of a Load-Exclusive/Store-Exclusive pair of synchronization primitives using, as examples, the LDREX and STREX instructions. The same description applies to any other pair of synchronization primitives:

- LDREXB used with STREXB
- LDREXH used with STREXH.

Each Load-Exclusive instruction must be used only with the corresponding Store-Exclusive instruction.

STREXD and LDREXD are not supported in ARMv7-M.

The model for the use of a Load-Exclusive/Store-Exclusive instruction pair, accessing memory address x is:

- The Load-Exclusive instruction always successfully reads a value from memory address x
- The corresponding Store-Exclusive instruction succeeds in writing back to memory address x only if no other processor or process has performed a more recent store of address x. The Store-Exclusive operation returns a status bit that indicates whether the memory write succeeded.

A Load-Exclusive instruction tags a small block of memory for exclusive access. The size of the tagged block is IMPLEMENTATION DEFINED, see *Tagging and the size of the tagged memory block* on page A3-15. A Store-Exclusive instruction to the same address clears the tag.

A3.4.1 Exclusive access instructions and Non-shareable memory regions

For memory regions that do not have the *Shareable* attribute, the exclusive access instructions rely on a *local monitor* that tags any address from which the processor executes a Load-Exclusive. Any non-aborted attempt by the same processor to use a Store-Exclusive to modify any address is guaranteed to clear the tag.

A Load-Exclusive performs a load from memory, and:

- the executing processor tags the physical memory address for exclusive access
- the local monitor of the executing processor transitions to its Exclusive Access state.

A Store-Exclusive performs a conditional store to memory, that depends on the state of the local monitor:

If the local monitor is in its Exclusive Access state

- If the address of the Store-Exclusive is the same as the address that has been tagged in the monitor by an earlier Load-Exclusive, then the store takes place, otherwise it is IMPLEMENTATION DEFINED whether the store takes place.
- A status value is returned to a register:
  - if the store took place the status value is 0
  - otherwise, the status value is 1.
- The local monitor of the executing processor transitions to its Open Access state.

If the local monitor is in its Open Access state

- no store takes place
- a status value of 1 is returned to a register.
- the local monitor remains in its Open Access state.

The Store-Exclusive instruction defines the register to which the status value is returned.

When a processor writes using any instruction other than a Store-Exclusive:

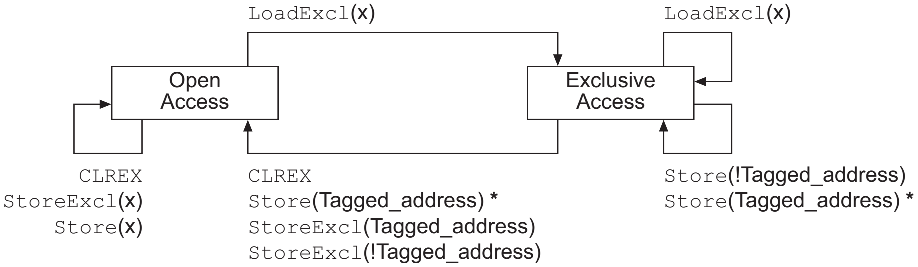
- if the write is to a physical address that is not covered by its local monitor the write does not affect the state of the local monitor
- if the write is to a physical address that is covered by its local monitor it is IMPLEMENTATION DEFINED whether the write affects the state of the local monitor.

If the local monitor is in its Exclusive Access state and a processor performs a Store-Exclusive to any address other than the last one from which it has performed a Load-Exclusive, it is IMPLEMENTATION DEFINED whether the store succeeds, but in all cases the local monitor is reset to its Open Access state. In ARMv7-M, the store must be treated as a software programming error.

Note

It is UNPREDICTABLE whether a store to a tagged physical address causes a tag in the local monitor to be cleared if that store is by an observer other than the one that caused the physical address to be tagged.

Figure A3-2 on page A3-10 shows the state machine for the local monitor. Table A3-6 on page A3-10 shows the effect of each of the operations shown in the figure.



Operations marked * are possible alternative IMPLEMENTATION DEFINED options.

In the diagram: LoadExcl represents any Load-Exclusive instruction
StoreExcl represents any Store-Exclusive instruction
Store represents any other store instruction.

Any LoadExcl operation updates the tagged address to the most significant bits of the address x used for the operation. For more information see the section *Size of the tagged memory block*.

Figure A3-2 Local monitor state machine diagram

Note

- The IMPLEMENTATION DEFINED options for the local monitor are consistent with the local monitor being constructed so that it does not hold any physical address, but instead treats any access as matching the address of the previous LDREX.
- A local monitor implementation can be unaware of Load-Exclusive and Store-Exclusive operations from other processors.
- It is UNPREDICTABLE whether the transition from Exclusive Access to Open Access state occurs when the STR or STREX is from another observer.

Table A3-6 shows the effect of the operations shown in Figure A3-2.

Table A3-6 Effect of Exclusive instructions and write operations on local monitor

Initial state	Operation[a]	Effect	Final state
Open Access	CLREX	No effect	Open Access
Open Access	StoreExcl(x)	Does not update memory, returns status 1	Open Access
Open Access	LoadExcl(x)	Loads value from memory, tags address x	Exclusive Access
Open Access	Store(x)	Updates memory, no effect on monitor	Open Access
Exclusive Access	CLREX	Clears tagged address	Open Access
Exclusive Access	StoreExcl(t)	Updates memory, returns status 0	Open Access

Table A3-6 Effect of Exclusive instructions and write operations on local monitor (continued)

Initial state	Operation[a]	Effect	Final state
Exclusive Access	`StoreExcl(!t)`	Updates memory, returns status 0[b]	Open Access
		Does not update memory, returns status 1[b]	
Exclusive Access	`LoadExcl(x)`	Loads value from memory, changes tag to address to x	Exclusive Access
Exclusive Access	`Store(!t)`	Updates memory, no effect on monitor	Exclusive Access
Exclusive Access	`Store(t)`	Updates memory	Exclusive Access[b]
			Open Access[b]

a. In the table:
`LoadExcl` represents any Load-Exclusive instruction
`StoreExcl` represents any Store-Exclusive instruction
`Store` represents any store operation other than a Store-Exclusive operation.
t is the tagged address, bits [31:a] of the address of the last Load-Exclusive instruction. For more information see *Tagging and the size of the tagged memory block* on page A3-15.

b. IMPLEMENTATION DEFINED alternative actions.

A3.4.2 Exclusive access instructions and Shareable memory regions

For memory regions that have the *Shareable* attribute, exclusive access instructions rely on:

- A *local monitor* for each processor in the system, that tags any address from which the processor executes a Load-Exclusive. The local monitor operates as described in *Exclusive access instructions and Non-shareable memory regions* on page A3-8, except that for Shareable memory, any Store-Exclusive described in that section as updating memory and/or returning the status value 0 is then subject to checking by the global monitor. The local monitor can ignore exclusive accesses from other processors in the system.

- A *global monitor* that tags a physical address as exclusive access for a particular processor. This tag is used later to determine whether a Store-Exclusive to the tagged address, that has not been failed by the local monitor, can occur. Any successful write to the tagged address by any other observer in the shareability domain of the memory location is guaranteed to clear the tag.

  For each processor in the system, the global monitor:

  — holds a single tagged address

  — maintains a state machine.

The global monitor can either reside in a processor block or exist as a secondary monitor at the memory interfaces.

An implementation can combine the functionality of the global and local monitors into a single unit.

Operation of the global monitor

Load-Exclusive from *Shareable* memory performs a load from memory, and causes the physical address of the access to be tagged as exclusive access for the requesting processor. This access also causes the exclusive access tag to be removed from any other physical address that has been tagged by the requesting processor. The global monitor only supports a single outstanding exclusive access to Shareable memory per processor.

Store-Exclusive performs a conditional store to memory:

- The store is guaranteed to succeed only if the physical address accessed is tagged as exclusive access for the requesting processor and both the local monitor and the global monitor state machines for the requesting processor are in the Exclusive Access state. In this case:
  - a status value of 0 is returned to a register to acknowledge the successful store
  - the final state of the global monitor state machine for the requesting processor is IMPLEMENTATION DEFINED
  - if the address accessed is tagged for exclusive access in the global monitor state machine for any other processor then that state machine transitions to Open Access state.
- If no address is tagged as exclusive access for the requesting processor, the store does not succeed:
  - a status value of 1 is returned to a register to indicate that the store failed
  - the global monitor is not affected and remains in Open Access state for the requesting processor.
- If a different physical address is tagged as exclusive access for the requesting processor, it is IMPLEMENTATION DEFINED whether the store succeeds or not:
  - if the store succeeds a status value of 0 is returned to a register, otherwise a value of 1 is returned
  - if the global monitor state machine for the processor was in the Exclusive Access state before the Store-Exclusive it is IMPLEMENTATION DEFINED whether that state machine transitions to the Open Access state.

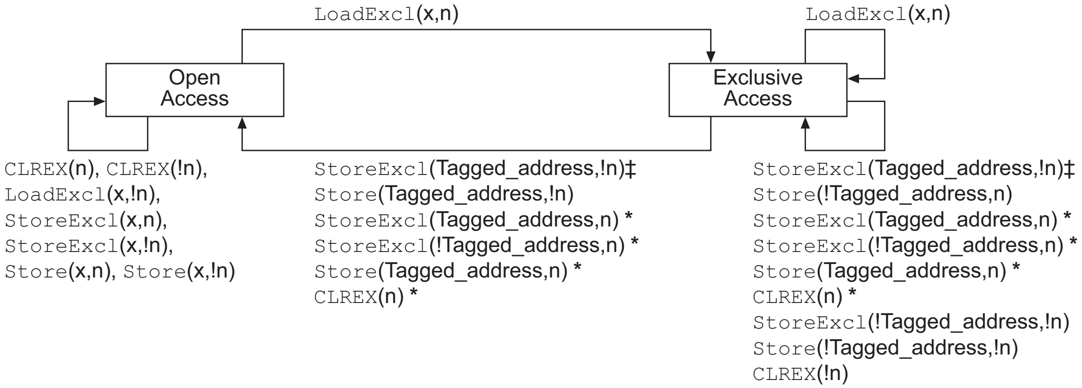
The Store-Exclusive instruction defines the register to which the status value is returned.

In a shared memory system, the global monitor implements a separate state machine for each processor in the system. The state machine for accesses to Shareable memory by processor (n) can respond to all the Shareable memory accesses visible to it. This means it responds to:

- accesses generated by the associated processor (n)
- accesses generated by the other observers in the shared memory system (!n).

In a shared memory system, the global monitor implements a separate state machine for each observer that can generate a Load-Exclusive or a Store-Exclusive in the system.

Figure A3-3 on page A3-13 shows the state machine for processor(n) in a global monitor. Table A3-7 on page A3-14 shows the effect of each of the operations shown in the figure.


LoadExcl(x,n)
LoadExcl(x,n)
Open Access
Exclusive Access
CLREX(n), CLREX(!n),
LoadExcl(x,!n),
StoreExcl(x,n),
StoreExcl(x,!n),
Store(x,n), Store(x,!n)
StoreExcl(Tagged_address,!n)‡
Store(Tagged_address,!n)
StoreExcl(Tagged_address,n) *
StoreExcl(!Tagged_address,n) *
Store(Tagged_address,n) *
CLREX(n) *
StoreExcl(Tagged_address,!n)‡
Store(!Tagged_address,n)
StoreExcl(Tagged_address,n) *
StoreExcl(!Tagged_address,n) *
Store(Tagged_address,n) *
CLREX(n) *
StoreExcl(!Tagged_address,!n)
Store(!Tagged_address,!n)
CLREX(!n)

‡ StoreExcl(Tagged_Address,!n) clears the monitor only if the StoreExcl updates memory

Operations marked * are possible alternative IMPLEMENTATION DEFINED options.

In the diagram: LoadExcl represents any Load-Exclusive instruction
StoreExcl represents any Store-Exclusive instruction
Store represents any other store instruction.

Any LoadExcl operation updates the tagged address to the most significant bits of the address x used for the operation. For more information see the section *Size of the tagged memory block*.

Figure A3-3 Global monitor state machine diagram for processor(n) in a multiprocessor system

Note

- Whether a Store-Exclusive successfully updates memory or not depends on whether the address accessed matches the tagged Shareable memory address for the processor issuing the Store-Exclusive instruction. For this reason, Figure A3-3 and Table A3-7 on page A3-14 only show how the (!n) entries cause state transitions of the state machine for processor(n).
- A Load-Exclusive can only update the tagged Shareable memory address for the processor issuing the Load-Exclusive instruction.
- The effect of the CLREX instruction on the global monitor is IMPLEMENTATION DEFINED.
- It is IMPLEMENTATION DEFINED whether a modification to a non-shareable memory location can cause a global monitor Exclusive Access to Open Access transition.
- It is IMPLEMENTATION DEFINED whether an LDREX to a non-shareable memory location can cause a global monitor Open Access to Exclusive Access transition.

Table A3-7 shows the effect of the operations shown in Figure A3-3 on page A3-13.

Table A3-7 Effect of load/store operations on global monitor for processor(n)

Initial state[a]	Operation[b]	Effect	Final state[a]
Open	`CLREX(n)`, `CLREX(!n)`	None	Open
Open	`StoreExcl(x,n)`	Does not update memory, returns status 1	Open
Open	`LoadExcl(x,!n)`	Loads value from memory, no effect on tag address for processor(n)	Open
Open	`StoreExcl(x,!n)`	Depends on state machine and tag address for processor issuing `STREX`[c]	Open
Open	`STR(x,n)`, `STR(x,!n)`	Updates memory, no effect on monitor	Open
Open	`LoadExcl(x,n)`	Loads value from memory, tags address x	Exclusive
Exclusive	`LoadExcl(x,n)`	Loads value from memory, tags address x	Exclusive
Exclusive	`CLREX(n)`	None. Effect on the final state is IMPLEMENTATION DEFINED.	Exclusive[e]
			Open[e]
Exclusive	`CLREX(!n)`	None	Exclusive
Exclusive	`StoreExcl(t,!n)`	Updates memory, returns status 0[c]	Open
		Does not update memory, returns status 1[c]	Exclusive
Exclusive	`StoreExcl(t,n)`	Updates memory, returns status 0[d]	Open
			Exclusive
Exclusive	`StoreExcl(!t,n)`	Updates memory, returns status 0[e]	Open
			Exclusive
		Does not update memory, returns status 1[e]	Open
			Exclusive
Exclusive	`StoreExcl(!t,!n)`	Depends on state machine and tag address for processor issuing `STREX`	Exclusive

Table A3-7 Effect of load/store operations on global monitor for processor(n) (continued)

Initial state[a]	Operation[b]	Effect	Final state[a]
Exclusive	`Store(t,n)`	Updates memory	Exclusive[e]
			Open[e]
Exclusive	`Store(t,!n)`	Updates memory	Open
Exclusive	`Store(!t,n)`, `Store(!t,!n)`	Updates memory, no effect on monitor	Exclusive

[a] Open = Open Access state, Exclusive = Exclusive Access state.

[b] In the table:

`LoadExcl` represents any Load-Exclusive instruction

`StoreExcl` represents any Store-Exclusive instruction

`Store` represents any store operation other than a Store-Exclusive operation.

t is the tagged address for processor(n), bits [31:a] of the address of the last Load-Exclusive instruction issued by processor(n), see *Tagging and the size of the tagged memory block*.

[c] The result of a `STREX(x,!n)` or a `STREX(t,!n)` operation depends on the state machine and tagged address for the processor issuing the `STREX` instruction. This table shows how each possible outcome affects the state machine for processor(n).

[d] After a successful `STREX` to the tagged address, the state of the state machine is IMPLEMENTATION DEFINED. However, this state has no effect on the subsequent operation of the global monitor.

[e] Effect is IMPLEMENTATION DEFINED. The table shows all permitted implementations.

A3.4.3 Tagging and the size of the tagged memory block

As shown in Figure A3-2 on page A3-10 and Figure A3-3 on page A3-13, when a `LDREX` instruction is executed, the resulting tag address ignores the least significant bits of the memory address:

```
Tagged_address == Memory_address[31:a]
```

The value of a in this assignment is IMPLEMENTATION DEFINED, between a minimum value of 2 and a maximum value of 11. For example, in an implementation where a = 4, a successful `LDREX` of address `0x000341B4` gives a tag value of bits [31:4] of the address, giving 0x000341B. This means that the four words of memory from `0x000341B0` to `0x000341BF` are tagged for exclusive access. Subsequently, a valid `STREX` to any address in this block will remove the tag.

The size of the tagged memory block is called the *Exclusives Reservation Granule*. The Exclusives Reservation Granule is IMPLEMENTATION DEFINED between:

- one word, in an implementation with a == 2
- 512 words, in an implementation with a == 11.

A3.4.4 Context switch support

It is necessary to ensure that the local monitor is in the Open Access state after a context switch. In ARMv7-M, the local monitor is changed to Open Access automatically as part of an exception entry or exit sequence. The local monitor can also be forced to the Open Access state by a CLREX instruction.

Note

Context switching is not an application level operation. However, this information is included here to complete the description of the exclusive operations.

A context switch might cause a subsequent Store-Exclusive to fail, requiring a load … store sequence to be replayed. To minimize the possibility of this happening, ARM recommends that the Store-Exclusive instruction is kept as close as possible to the associated Load-Exclusive instruction, see *Load-Exclusive and Store-Exclusive usage restrictions*.

A3.4.5 Load-Exclusive and Store-Exclusive usage restrictions

The Load-Exclusive and Store-Exclusive instructions are designed to work together, as a pair, for example a LDREX/STREX pair or a LDREXB/STREXB pair. As mentioned in *Context switch support*, ARM recommends that the Store-Exclusive instruction always follows within a few instructions of its associated Load-Exclusive instructions. In order to support different implementations of these functions, software must follow the notes and restrictions given here.

These notes describe use of a LDREX/STREX pair, but apply equally to any other Load-Exclusive/Store-Exclusive pair:

- The exclusives support a single outstanding exclusive access for each processor thread that is executed. The architecture makes use of this by not requiring an address or size check as part of the IsExclusiveLocal() function. If the target address of an STREX is different from the preceding LDREX in the same execution thread, behavior can be UNPREDICTABLE. As a result, an LDREX/STREX pair can only be relied upon to eventually succeed if they are executed with the same address.
- An explicit store to memory can cause the clearing of exclusive monitors associated with other processors, therefore, performing a store between the LDREX and the STREX can result in a livelock situation. As a result, code must avoid placing an explicit store between an LDREX and an STREX in a single code sequence.
- If two STREX instructions are executed without an intervening LDREX the second STREX returns a status value of 1. This means that:
  - every STREX must have a preceding LDREX associated with it in a given thread of execution
  - it is not necessary for every LDREX to have a subsequent STREX.
- An implementation of the Load-Exclusive and Store-Exclusive instructions can require that, in any thread of execution, the transaction size of a Store-Exclusive is the same as the transaction size of the preceding Load-Exclusive that was executed in that thread. If the transaction size of a Store-Exclusive is different from the preceding Load-Exclusive in the same execution thread, behavior can be UNPREDICTABLE. As a result, software can rely on a Load-Exclusive/Store-Exclusive pair to eventually succeed only if they are executed with the same address.

- An implementation might clear an exclusive monitor between the LDREX and the STREX, without any application-related cause. For example, this might happen because of cache evictions. Code written for such an implementation must avoid having any explicit memory accesses or cache maintenance operations between the LDREX and STREX instructions.
- Implementations can benefit from keeping the LDREX and STREX operations close together in a single code sequence. This minimizes the likelihood of the exclusive monitor state being cleared between the LDREX instruction and the STREX instruction. Therefore, ARM recommends strongly a limit of 128 bytes between LDREX and STREX instructions in a single code sequence, for best performance.
- Implementations that implement coherent protocols, or have only a single master, might combine the local and global monitors for a given processor. The IMPLEMENTATION DEFINED and UNPREDICTABLE parts of the definitions in *Pseudocode details of operations on exclusive monitors* on page B2-8 are provided to cover this behavior.
- The architecture sets an upper limit of 2048 bytes on the size of a region that can be marked as exclusive. Therefore, for performance reasons, ARM recommends that software separates objects that will be accessed by exclusive accesses by at least 2048 bytes. This is a performance guideline rather than a functional requirement.
- LDREX and STREX operations must be performed only on memory with the Normal memory attribute.
- If the memory attributes for the memory being accessed by an LDREX/STREX pair are changed between the LDREX and the STREX, behavior is UNPREDICTABLE.

A3.4.6 Synchronization primitives and the memory order model

The synchronization primitives follow the memory ordering model of the memory type accessed by the instructions. For this reason:

- Portable code for claiming a spinlock must include a DMB instruction between claiming the spinlock and making any access that makes use of the spinlock.
- Portable code for releasing a spinlock must include a DMB instruction before writing to clear the spinlock.

This requirement applies to code using the Load-Exclusive/Store-Exclusive instruction pairs, for example LDREX/STREX.

A3.5 Memory types and attributes and the memory order model

ARMv7 defines a set of memory attributes with the characteristics required to support the memory and devices in the system memory map.

The ordering of accesses for regions of memory, referred to as the memory order model, is defined by the memory attributes. This model is described in the following sections:

- *Memory types*
- *Summary of ARMv7 memory attributes* on page A3-19
- *Atomicity in the ARM architecture* on page A3-20
- *Normal memory* on page A3-22
- *Device memory* on page A3-24
- *Strongly-ordered memory* on page A3-25
- *Memory access restrictions* on page A3-26

A3.5.1 Memory types

For each memory region, the most significant memory attribute specifies the memory type. There are three mutually exclusive memory types:

- Normal
- Device
- Strongly-ordered.

Normal and Device memory regions have additional attributes.

Usually, memory used for program code and for data storage is Normal memory. Examples of Normal memory technologies are:

- programmed Flash ROM

  Note

  During programming, Flash memory can be ordered more strictly than Normal memory.

- ROM
- SRAM
- DRAM and DDR memory.

System peripherals (I/O) generally conform to different access rules to Normal memory. Examples of I/O accesses are:

- FIFOs where consecutive accesses
  - — add queued values on write accesses
  - — remove queued values on read accesses.
- interrupt controller registers where an access can be used as an interrupt acknowledge, changing the state of the controller itself

- memory controller configuration registers that are used to set up the timing and correctness of areas of Normal memory
- memory-mapped peripherals, where accessing a memory location can cause side effects in the system.

In ARMv7, regions of the memory map for these accesses are defined as Device or Strongly-ordered memory. To ensure system correctness, access rules for Device and Strongly-ordered memory are more restrictive than those for Normal memory:

- both read and write accesses can have side effects
- accesses must not be repeated, for example, on return from an exception
- the number, order and sizes of the accesses must be maintained.

In addition, for Strongly-ordered memory, all memory accesses are strictly ordered to correspond to the program order of the memory access instructions.

A3.5.2 Summary of ARMv7 memory attributes

Table A3-8 summarizes the memory attributes. For more information about theses attributes see:

- *Normal memory* on page A3-22 and *Shareable attribute for Device memory regions* on page A3-25, for the *shareability* attribute
- *Write-Through cacheable, Write-Back cacheable and Non-cacheable Normal memory* on page A3-23, for the *cacheability* attribute.

Table A3-8 Memory attribute summary

Memory type attribute	Shareability	Other attributes	Description
Strongly-ordered	-		All memory accesses to Strongly-ordered memory occur in program order. All Strongly-ordered regions are assumed to be Shareable.
Device	Shareable		Intended to handle memory-mapped peripherals that are shared by several processors.
	Non-shareable		Intended to handle memory-mapped peripherals that are used only by a single processor.

Table A3-8 Memory attribute summary (continued)

Memory type attribute	Shareability	Other attributes	Description
Normal	Shareable	Cacheability, one of: [a] Non-cacheable Write-Through cacheable Write-Back Write-Allocate cacheable Write-Back no Write-Allocate cacheable	Intended to handle Normal memory that is shared between several processors.
	Non-shareable	Cacheability, one of: [a] Non-cacheable Write-Through cacheable Write-Back Write-Allocate cacheable Write-Back no Write-Allocate cacheable	Intended to handle Normal memory that is used by only a single processor.

a. The cacheability attribute is defined independently for inner and outer cache regions.

A3.5.3 Atomicity in the ARM architecture

Atomicity is a feature of memory accesses, described as *atomic* accesses. The ARM architecture description refers to two types of atomicity, defined in:

- *Single-copy atomicity*
- *Multi-copy atomicity* on page A3-21.

Single-copy atomicity

A read or write operation is *single-copy atomic* if the following conditions are both true:

- After any number of write operations to an operand, the value of the operand is the value written by one of the write operations. It is impossible for part of the value of the operand to come from one write operation and another part of the value to come from a different write operation.
- When a read operation and a write operation are made to the same operand, the value obtained by the read operation is one of:
  - — the value of the operand before the write operation
  - — the value of the operand after the write operation.

  It is never the case that the value of the read operation is partly the value of the operand before the write operation and partly the value of the operand after the write operation.

In ARMv7-M, the single-copy atomic processor accesses are:

- all byte accesses
- all halfword accesses to halfword-aligned locations
- all word accesses to word-aligned locations

LDM, LDC, LDC2, LDRD, STM, STC, STC2, STRD, PUSH, and POP instructions are executed as a sequence of word-aligned word accesses. Each 32-bit word access is guaranteed to be single-copy atomic. A subsequence of two or more word accesses from the sequence might not exhibit single-copy atomicity.

When an access is not single-copy atomic, it is executed as a sequence of smaller accesses, each of which is single-copy atomic, at least at the byte level.

If an instruction is executed as a sequence of accesses according to these rules, some exceptions can be taken in the sequence and cause execution of the instruction to be abandoned.

On exception return, the instruction that generated the sequence of accesses is re-executed and so any accesses that had already been performed before the exception was taken might be repeated, see *Exceptions in LDM and STM operations* on page B1-30.

Note

The exception behavior for these multiple access instructions means they are not suitable for use for writes to memory for the purpose of software synchronization.

For implicit accesses:

- Cache linefills and evictions have no effect on the single-copy atomicity of explicit transactions or instruction fetches.
- Instruction fetches are single-copy atomic for each instruction fetched.

  Note

  32-bit Thumb instructions are fetched as two 16-bit items.

Multi-copy atomicity

In a multiprocessing system, writes to a memory location are *multi-copy atomic* if the following conditions are both true:

- All writes to the same location are *serialized*, meaning they are observed in the same order by all observers, although some observers might not observe all of the writes.
- A read of a location does not return the value of a write until all observers observe that write.

Writes to Normal memory are not multi-copy atomic.

All writes to Device and Strongly-Ordered memory that are single-copy atomic are also multi-copy atomic.

All write accesses to the same location are serialized. Write accesses to Normal memory can be repeated up to the point that another write to the same address is observed.

For Normal memory, serialization of writes does not prohibit the merging of writes.

A3.5.4 Normal memory

Normal memory is idempotent, meaning that it exhibits the following properties:

- read accesses can be repeated with no side effects
- repeated read accesses return the last value written to the resource being read
- read accesses can prefetch additional memory locations with no side effects
- write accesses can be repeated with no side effects, provided that the contents of the location are unchanged between the repeated writes
- unaligned accesses can be supported
- accesses can be merged before accessing the target memory system.

Normal memory can be read/write or read-only, and a Normal memory region is defined as being either Shareable or Non-shareable.

The Normal memory type attribute applies to most memory used in a system.

Accesses to Normal memory have a weakly consistent model of memory ordering. See a standard text describing memory ordering issues for a description of weakly consistent memory models, for example chapter 2 of *Memory Consistency Models for Shared Memory-Multiprocessors*, Kourosh Gharachorloo, Stanford University Technical Report CSL-TR-95-685. In general, for Normal memory, barrier operations are required where the order of memory accesses observed by other observers must be controlled. This requirement applies regardless of the cacheability and shareability attributes of the Normal memory region.

The ordering requirements of accesses described in *Ordering requirements for memory accesses* on page A3-32 apply to all explicit accesses.

An instruction that generates a sequence of accesses as described in *Atomicity in the ARM architecture* on page A3-20 might be abandoned as a result of an exception being taken during the sequence of accesses. On return from the exception the instruction is restarted, and therefore one or more of the memory locations might be accessed multiple times. This can result in repeated write accesses to a location that has been changed between the write accesses.

Note

For ARMv7-M, the `LDM` and `STM` instructions can restart or continue on exception return, see *Exceptions in LDM and STM operations* on page B1-30.

Non-shareable Normal memory

For a Normal memory region, the Non-shareable attribute identifies Normal memory that is likely to be accessed only by a single processor.

A region of memory marked as Non-shareable Normal does not have any requirement to make the effect of a cache transparent for data or instruction accesses. If other observers share the memory system, software must use cache maintenance operations if the presence of caches might lead to coherency issues when communicating between the observers. This cache maintenance requirement is in addition to the barrier operations that are required to ensure memory ordering.

For Non-shareable Normal memory, the Load Exclusive and Store Exclusive synchronization primitives do not take account of the possibility of accesses by more than one observer.

Shareable Normal memory

For Normal memory, the Shareable memory attribute describes Normal memory that is expected to be accessed by multiple processors or other system masters.

A region of Normal memory with the Sharable attribute is one for which the effect of interposing a cache, or caches, on the memory system is entirely transparent to data accesses in the same shareability domain. Explicit software management is needed to ensure the coherency of instruction caches.

Implementations can use a variety of mechanisms to support this management requirement, from simply not caching accesses in Shareable regions to more complex hardware schemes for cache coherency for those regions.

For Shareable Normal memory, the Load-Exclusive and Store-Exclusive synchronization primitives take account of the possibility of accesses by more than one observer in the same Shareability domain.

Note

The Shareable concept enables system designers to specify the locations in Normal memory that must have coherency requirements. However, to facilitate porting of software, software developers must not assume that specifying a memory region as Non-shareable permits software to make assumptions about the incoherency of memory locations between different processors in a shared memory system. Such assumptions are not portable between different multiprocessing implementations that make use of the Shareable concept. Any multiprocessing implementation might implement caches that, inherently, are shared between different processing elements.

Write-Through cacheable, Write-Back cacheable and Non-cacheable Normal memory

In addition to being Shareable or Non-shareable, each region of Normal memory can be marked as being one of:

- Write-Through cacheable

- Write-Back cacheable, with an additional qualifier that marks it as one of:
  - Write-Back, Write-Allocate
  - Write-Back, no Write-Allocate
- Non-cacheable.

The cacheability attributes for a region are independent of the shareability attributes for the region. The cacheability attributes indicate the required handling of the data region if it is used for purposes other than the handling of shared data. This independence means that, for example, a region of memory that is marked as being cacheable and Shareable might not be held in the cache in an implementation where Shareable regions do not cache their data.

A3.5.5 Device memory

The Device memory type attribute defines memory locations where an access to the location can cause side effects, or where the value returned for a load can vary depending on the number of loads performed. memory-mapped peripherals and I/O locations are examples of memory regions that normally are marked as being Device.

For explicit accesses from the processor to memory marked as Device:

- all accesses occur at their program size
- the number of accesses is the number specified by the program.

An implementation must not repeat an access to a Device memory location if the program has only one access to that location. In other words, accesses to Device memory locations are not restartable.

The architecture does not permit speculative accesses to memory marked as Device.

Address locations marked as Device are Non-cacheable. While writes to Device memory can be buffered, writes can be merged only where the merge maintains:

- the number of accesses
- the order of the accesses
- the size of each access.

Multiple accesses to the same address must not change the number of accesses to that address. Coalescing of accesses is not permitted for accesses to Device memory.

When a Device memory operation has side effects that apply to Normal memory regions, software must use a Memory Barrier to ensure correct execution. An example is programming the configuration registers of a memory controller with respect to the memory accesses it controls.

All explicit accesses to Device memory must comply with the ordering requirements of accesses described in *Ordering requirements for memory accesses* on page A3-32.

An instruction that generates a sequence of accesses as described in *Atomicity in the ARM architecture* on page A3-20 might be abandoned as a result of an exception being taken during the sequence of accesses. On return from the exception the instruction is restarted, and therefore one or more of the memory locations might be accessed multiple times. This can result in repeated write accesses to a location that has been changed between the write accesses.

Note

Do not use an instruction that generates a sequence of accesses to access Device memory if the instruction might restart after an exception and repeat any write accesses, see *Exceptions in LDM and STM operations* on page B1-30 for more information.

Any unaligned access that is not faulted by the alignment restrictions and accesses Device memory has UNPREDICTABLE behavior.

Shareable attribute for Device memory regions

Device memory regions can be given the Shareable attribute. This means that a region of Device memory can be described as either:

- Shareable Device memory
- Non-shareable Device memory.

Non-shareable Device memory is defined as only accessible by a single processor. An example of a system supporting Shareable and Non-shareable Device memory is an implementation that supports both:

- a local bus for its private peripherals
- system peripherals implemented on the main shared system bus.

Such a system might have more predictable access times for local peripherals such as watchdog timers or interrupt controllers. In particular, a specific address in a Non-shareable Device memory region might access a different physical peripheral for each processor.

A3.5.6 Strongly-ordered memory

~~Memory regions with the Strongly-ordered memory type attribute have a strong memory-ordering model for all explicit memory accesses from a processor. Any access to memory with the Strongly-ordered attribute must act as if `DMB UN` instructions were inserted before and after the access from the processor. See *Data Memory Barrier (DMB)* on page A3-35.~~

When synchronization is required, a program must include an explicit Memory Barrier between the memory access and the following instruction, see *Data Synchronization Barrier (DSB)* on page A3-36.

For explicit accesses from the processor to memory marked as Strongly-ordered:

- all accesses occur at their program size
- the number of accesses is the number specified by the program.

~~An implementation must not repeat an access to a Strongly-ordered memory location if the program has only one access to that location. In other words, accesses to Strongly-ordered memory locations are not restartable.~~

The architecture does not permit speculative accesses to memory marked as Strongly-ordered.

Address locations in Strongly-ordered memory are not held in a cache, and are always treated as Shareable memory locations.

All explicit accesses to Strongly-ordered memory must correspond to the ordering requirements of accesses described in *Ordering requirements for memory accesses* on page A3-32.

An instruction that generates a sequence of accesses as described in *Atomicity in the ARM architecture* on page A3-20 might be abandoned as a result of an exception being taken during the sequence of accesses. On return from the exception the instruction is restarted, and therefore one or more of the memory locations might be accessed multiple times. This can result in repeated write accesses to a location that has been changed between the write accesses.

Note

Do not use an instruction that generates a sequence of accesses to access Strongly-ordered memory if the instruction might restart after an exception and repeat any write accesses, see *Exceptions in LDM and STM operations* on page B1-30 for more information.

Any unaligned access that is not faulted by the alignment restrictions and accesses Strongly-ordered memory has UNPREDICTABLE behavior.

A3.5.7 Memory access restrictions

The following restrictions apply to memory accesses:

- For any access X, the bytes accessed by X must all have the same memory type attribute, otherwise the behavior of the access is UNPREDICTABLE. That is, an unaligned access that spans a boundary between different memory types is UNPREDICTABLE.

- For any two memory accesses X and Y that are generated by the same instruction, the bytes accessed by X and Y must all have the same memory type attribute, otherwise the results are UNPREDICTABLE. For example, an `LDC`, `LDM`, `LDRD`, `STC`, `STM`, or `STRD` that spans a boundary between Normal and Device memory is UNPREDICTABLE.

- An instruction that generates an unaligned memory access to Device or Strongly-ordered memory is UNPREDICTABLE.

- For instructions that generate accesses to Device or Strongly-ordered memory, implementations must not change the sequence of accesses specified by the pseudocode of the instruction. This includes not changing:
  - — how many accesses there are
  - — the time order of the accesses
  - — the data sizes and other properties of each access.

  In addition, processor core implementations expect any attached memory system to be able to identify the memory type of an accesses, and to obey similar restrictions with regard to the number, time order, data sizes and other properties of the accesses.

Exceptions to this rule are:

- — An implementation of a processor core can break this rule, provided that the information it supplies to the memory system enables the original number, time order, and other details of the accesses to be reconstructed. In addition, the implementation must place a requirement on attached memory systems to do this reconstruction when the accesses are to Device or Strongly-ordered memory.

  For example, an implementation with a 64-bit bus might pair the word loads generated by an `LDM` into 64-bit accesses. This is because the instruction semantics ensure that the 64-bit access is always a word load from the lower address followed by a word load from the higher address. However the implementation must permit the memory systems to unpack the two word loads when the access is to Device or Strongly-ordered memory.

- — Any implementation technique that produces results that cannot be observed to be different from those described above is legitimate.

- `LDM` and `STM` instructions that are used with the `IT` instruction are restartable if interrupted during execution. Restarting a load or store instruction is incompatible with the Device and Strongly Ordered memory access rules. For details on the architecture constraints associated with LDM and STM and the exception model see *Exceptions in LDM and STM operations* on page B1-30.

- Any multi-access instruction that loads or stores the PC must access only Normal memory. If the instruction accesses Device or Strongly-ordered memory the result is UNPREDICTABLE.

- Any instruction fetch must access only Normal memory. If it accesses Device or Strongly-ordered memory, the result is UNPREDICTABLE. For example, instruction fetches must not be performed to an area of memory that contains read-sensitive devices, because there is no ordering requirement between instruction fetches and explicit accesses.

To ensure correctness, read-sensitive locations must be marked as non-executable (see *Privilege level access controls for instruction accesses* on page A3-28).

A3.6 Access rights

ARMv7 includes additional attributes for memory regions, that enable:

- Data accesses to be restricted, based on the privilege of the access. See *Privilege level access controls for data accesses*.
- Instruction fetches to be restricted, based on the privilege of the process or thread making the fetch. See *Privilege level access controls for instruction accesses*.

A3.6.1 Privilege level access controls for data accesses

The memory attributes can define that a memory region is:

- not accessible to any accesses
- accessible only to Privileged accesses
- accessible to Privileged and Unprivileged accesses.

The access privilege level is defined separately for explicit read and explicit write accesses. However, a system that defines the memory attributes is not required to support all combinations of memory attributes for read and write accesses.

A Privileged access is an access made during privileged execution, as a result of a load or store operation other than `LDRT`, `STRT`, `LDRBT`, `STRBT`, `LDRHT`, `STRHT`, `LDRSHT`, or `LDRSBT`.

An Unprivileged access is an access made as a result of load or store operation performed in one of these cases:

- when the current execution mode is configured for Unprivileged access only
- when the processor is in any mode and the access is made as a result of a `LDRT`, `STRT`, `LDRBT`, `STRBT`, `LDRHT`, `STRHT`, `LDRSHT`, or `LDRSBT` instruction.

An exception occurs if the processor attempts a data access that the access rights do not permit. For example, a MemManage exception occurs if the processor mode is Unprivileged and the processor attempts to access a memory region that is marked as only accessible to Privileged accesses.

Note

Data access control is only supported when a *Memory Protection Unit* is implemented and enabled, see *Protected Memory System Architecture (PMSAv7)* on page B3-35.

A3.6.2 Privilege level access controls for instruction accesses

Memory attributes can define that a memory region is:

- not accessible for execution
- accessible for execution by Privileged processes only
- accessible for execution by Privileged and Unprivileged processes.

To define the instruction access rights to a memory region, the memory attributes describe, separately, for the region:

- its read access rights
- whether it is *suitable for execution*.

For example, a region that is accessible for execution by Privileged processes has the memory attributes:

- accessible only to Privileged read accesses
- suitable for execution.

This means there is some linkage between the memory attributes that define the accessibility of a region to explicit memory accesses, and those that define that a region can be executed.

A MemManage exception occurs if a processor attempts to execute code from a memory location with attributes that do not permit code execution.

Note

Instruction access control is fully supported when a *Memory Protection Unit* is implemented and enabled, see *Protected Memory System Architecture (PMSAv7)* on page B3-35.

Instruction execution access control is also supported in the default address map, see *The system address map* on page B3-2.

A3.7 Memory access order

ARMv7 provides a set of three memory types, Normal, Device, and Strongly-ordered, with well-defined memory access properties.

The ARMv7 application-level view of the memory attributes is described in:

- *Memory types and attributes and the memory order model* on page A3-18
- *Access rights* on page A3-28.

When considering memory access ordering, an important feature is the *Shareable* memory attribute that indicates whether a region of memory can be shared between multiple processors, and therefore requires an appearance of cache transparency in the ordering model.

The key issues with the memory order model depend on the target audience:

- For software programmers, considering the model at the application level, the key factor is that for accesses to Normal memory, barriers are required in some situations where the order of accesses observed by other observers must be controlled.
- For silicon implementers, considering the model at the system level, the Strongly-ordered and Device memory attributes place certain restrictions on the system designer in terms of what can be built and when to indicate completion of an access.

  Note

  Implementations remain free to choose the mechanisms required to implement the functionality of the memory model.

More information about the memory order model is given in the following subsections:

- *Reads and writes*
- *Ordering requirements for memory accesses* on page A3-32
- *Memory barriers* on page A3-35.

Additional attributes and behaviors relate to the memory system architecture. These features are defined in the system level section of this manual, see *Protected Memory System Architecture (PMSAv7)* on page B3-35.

A3.7.1 Reads and writes

Each memory access is either a read or a write. *Explicit* memory accesses are the memory accesses required by the function of an instruction. The following can cause memory accesses that are not explicit:

- instruction fetches
- cache loads and writebacks

Except where otherwise stated, the memory ordering requirements only apply to explicit memory accesses.

Reads

Reads are defined as memory operations that have the semantics of a load.

The memory accesses of the following instructions are reads:

- LDR, LDRB, LDRH, LDRSB, and LDRSH
- LDRT, LDRBT, LDRHT, LDRSBT, and LDRSHT
- LDREX, LDREXB, and LDREXH
- LDM{IA,DB}, LDRD, and POP
- LDC and LDC2
- the return of status values by STREX, STREXB, and STREXH
- TBB and TBH.

Writes

Writes are defined as memory operations that have the semantics of a store.

The memory accesses of the following instructions are Writes:

- STR, STRB, and STRH
- STRT, STRBT, and STRHT
- STREX, STREXB, and STREXH
- STM{IA,DB}, STRD, and PUSH
- STC and STC2

Synchronization primitives

Synchronization primitives must ensure correct operation of system semaphores in the memory order model. The synchronization primitive instructions are defined as those instructions that are used to ensure memory synchronization:

- LDREX, STREX, LDREXB, STREXB, LDREXH, STREXH.

For details of the Load-Exclusive, Store-Exclusive and Clear-Exclusive instructions see *Synchronization and semaphores* on page A3-8.

The Load-Exclusive and Store-Exclusive instructions are supported to Shareable and Non-shareable memory. Non-shareable memory can be used to synchronize processes that are running on the same processor. Shareable memory must be used to synchronize processes that might be running on different processors.

Observability and completion

The set of observers that can observe a memory access is defined by the system.

For all memory:

- a write to a location in memory is said to be observed by an observer when a subsequent read of the location by the same observer will return the value written by the write

- a write to a location in memory is said to be globally observed for a shareability domain when a subsequent read of the location by any observer within that shareability domain that is capable of observing the write will return the value written by the write
- a read of a location in memory is said to be observed by an observer when a subsequent write to the location by the same observer will have no effect on the value returned by the read
- a read of a location in memory is said to be globally observed for a shareability domain when a subsequent write to the location by any observer within that shareability domain that is capable of observing the write will have no effect on the value returned by the read.

Additionally, for Strongly-ordered memory:

- A read or write of a memory-mapped location in a peripheral that exhibits side-effects is said to be observed, and globally observed, only when the read or write:
  - meets the general conditions listed
  - can begin to affect the state of the memory-mapped peripheral
  - can trigger all associated side effects, whether they affect other peripheral devices, cores or memory.

For all memory, the ARMv7-M completion rules are defined as:

- A read or write is complete for a shareability domain when all of the following are true:
  - the read or write is globally observed for that shareability domain
  - any instruction fetches by observers within the shareability domain have observed the read or write.
- A cache or branch predictor maintenance operation is complete for a shareability domain when the effects of operation are globally observed for that shareability domain.

Side effect completion in Strongly-ordered and Device memory

The completion of a memory access in Strongly-ordered or Device memory is not guaranteed to be sufficient to determine that the side effects of the memory access are visible to all observers. The mechanism that ensures the visibility of side-effects of a memory access is IMPLEMENTATION DEFINED, for example provision of a status register that can be polled.

A3.7.2 Ordering requirements for memory accesses

ARMv7-M defines access restrictions in the permitted ordering of memory accesses. These restrictions depend on the memory attributes of the accesses involved.

Two terms used in describing the memory access ordering requirements are:

Address dependency

An address dependency exists when the value returned by a read access is used to compute the address of a subsequent read or write access. An address dependency exists even if the value read by the first read access does not change the address of the second read or write access. This might be the case if the value returned is masked off before it is used, or if it has no effect on the predicted address value for the second access.

Control dependency

A control dependency exists when the data value returned by a read access is used to determine the condition code flags, and the values of the flags are used for condition code evaluation to determine the address of a subsequent read access. This address determination might be through conditional execution, or through the evaluation of a branch

Figure A3-4 on page A3-34 shows the memory ordering between two explicit accesses A1 and A2, where A1 occurs before A2 in program order. The symbols used in the figure are as follows:

< Accesses must be globally observed in program order, that is, A1 must be globally observed strictly before A2.

- Accesses can be globally observed in any order, provided that the requirements of uniprocessor semantics, for example respecting dependencies between instructions in a single processor, are maintained.

The following additional restrictions apply to the ordering of memory accesses that have this symbol:

- If there is an address dependency then the two memory accesses are observed in program order.

  This ordering restriction does not apply if there is only a control dependency between the two read accesses.

  If there is both an address dependency and a control dependency between two read accesses the ordering requirements of the address dependency apply.
- If the value returned by a read access is used as data written by a subsequent write access, then the two memory accesses are observed in program order.
- It is impossible for an observer to observe a write access to a memory location if that location would not be written to in a sequential execution of a program
- It is impossible for an observer to observe a write value to a memory location if that value would not be written in a sequential execution of a program.

In Figure A3-4 on page A3-34, an access refers to a read or a write access to the specified memory type. For example, *Device access, Non-shareable* refers to a read or write access to Non-shareable Device memory.

A2 A1	Normal access	Device access		Strongly Ordered access
		Non-shareable	Shareable	
Normal access	-	-	-	<
Device access, Non-shareable	-	<	-	<
Device access, Shareable	-	-	<	<
Strongly Ordered access	<	<	<	<

Figure A3-4 Memory ordering restrictions

There are no ordering requirements for implicit accesses to any type of memory.

Program order for instruction execution

The program order of instruction execution is the order of the instructions in the control flow trace.

Explicit memory accesses in an execution can be either:

Strictly Ordered
Denoted by <. Must occur strictly in order.

Ordered Denoted by <=. Can occur either in order or simultaneously.

Multiple load and store instructions, LDC, LDC2, LDMDB, LDMIA, LDRD, POP, PUSH, STC, STC2, STMDB, STMIA, and STRD, generate multiple word accesses, each of which is a separate access for the purpose of determining ordering.

The rules for determining program order for two accesses A1 and A2 are:

If A1 and A2 are generated by two different instructions:

- A1 < A2 if the instruction that generates A1 occurs before the instruction that generates A2 in program order
- A2 < A1 if the instruction that generates A2 occurs before the instruction that generates A1 in program order.

If A1 and A2 are generated by the same instruction:

- If A1 and A2 are two word loads generated by an LDC, LDC2, LDMDB, LDMIA or POP instruction, or two word stores generated by a PUSH, STC, STC2, STMDB, or STMIA instruction, excluding LDMDB, LDMIA or POP instructions with a register list that includes the PC:
  - A1 <= A2 if the address of A1 is less than the address of A2
  - A2 <= A1 if the address of A2 is less than the address of A1.
- If A1 and A2 are two word loads generated by an LDMDB, LDMIA or POP instruction with a register list that includes the PC, the program order of the memory accesses is not defined.
- If A1 and A2 are two word loads generated by an LDRD instruction or two word stores generated by an STRD instruction, the program order of the memory accesses is not defined.

- For any instruction or operation not explicitly mentioned in this section, if the single-copy atomicity rules described in *Single-copy atomicity* on page A3-20 mean the operation becomes a sequence of accesses, then the time-ordering of those accesses is not defined.

A3.7.3 Memory barriers

Memory barrier is the general term applied to an instruction, or sequence of instructions, used to force synchronization events by a processor with respect to retiring load and store instructions in a processor core. A memory barrier is used to guarantee both:

- completion of preceding load and store instructions to the programmers' model
- flushing of any prefetched instructions before the memory barrier event.

ARMv7-M requires three explicit memory barriers to support the memory order model described in this chapter. The three memory barriers are:

- Data Memory Barrier, see *Data Memory Barrier (DMB)*
- Data Synchronization Barrier, see *Data Synchronization Barrier (DSB)* on page A3-36
- Instruction Synchronization Barrier, see *Instruction Synchronization Barrier (ISB)* on page A3-37.

The DMB and DSB memory barriers affect reads and writes to the memory system generated by load and store instructions. Instruction fetches are not explicit accesses and are not affected.

Note

In ARMv7-M, memory barrier operations might be required in conjunction with data or unified cache and branch predictor maintenance operations.

Data Memory Barrier (DMB)

The `DMB` instruction is a data memory barrier. The processor that executes the `DMB` instruction is referred to as the *executing processor*, Pe. The `DMB` instruction takes the *required shareability domain* and *required access types* as arguments.

Note

ARMv7-M only supports system-wide barriers with no shareability domain or access type limitations.

A DMB creates two groups of memory accesses, *Group A* and *Group B*:

Group A Contains:

- all explicit memory accesses of the required access types from observers within the same shareability domain as Pe that are observed by Pe before the `DMB` instruction. This includes any accesses of the required access types and required shareability domain performed by Pe.

- all loads of required access types from observers within the same shareability domain as Pe that have been observed by any given observer Py within the same required shareability domain as Pe before Py has performed a memory access that is a member of Group A.

Group B Contains:

- all explicit memory accesses of the required access types by Pe that occur in program order after the `DMB` instruction
- all explicit memory accesses of the required access types by any given observer Px within the same required shareability domain as Pe that can only occur after Px has observed a store that is a member of Group B.

Any observer with the same required shareability domain as Pe observes all members of Group A before it observes any member of Group B. Where members of Group A and Group B access the same memory-mapped peripheral, all members of Group A will be visible at the memory-mapped peripheral before any members of Group B are visible at that peripheral.

Note

- A memory access might be in neither Group A nor Group B. The DMB does not affect the order of observation of such a memory access.
- The second part of the definition of Group A is recursive. Ultimately, membership of Group A derives from the observation by Py of a load before Py performs an access that is a member of Group A as a result of the first part of the definition of Group A.
- The second part of the definition of Group B is recursive. Ultimately, membership of Group B derives from the observation by any observer of an access by Pe that is a member of Group B as a result of the first part of the definition of Group B.

`DMB` only affects memory accesses. It has no effect on the ordering of any other instructions executing on the processor.

For details of the `DMB` instruction see *DMB* on page A6-68.

Data Synchronization Barrier (DSB)

The `DSB` instruction is a special memory barrier, that synchronizes the execution stream with memory accesses. The `DSB` instruction takes the *required shareability domain* and *required access types* as arguments. A `DSB` behaves as a `DMB` with the same arguments, and also has the additional properties defined here.

Note

ARMv7-M only supports system-wide barriers with no shareability domain or access type limitations.

A DSB completes when both:

- all explicit memory accesses that are observed by Pe before the DSB is executed, are of the required access types, and are from observers in the same required shareability domain as Pe, are complete for the set of observers within the required shareability domain
- all Cache and Branch predictor maintenance operations issued by Pe before the DSB are complete.

In addition, no instruction that appears in program order after the DSB instruction can execute until the DSB completes.

For details of the DSB instruction see *DSB* on page A6-70.

Instruction Synchronization Barrier (ISB)

An ISB instruction flushes the pipeline in the processor, so that all instructions that come after the ISB instruction in program order are fetched from cache or memory only after the ISB instruction has completed. Using an ISB ensures that the effects of context altering operations executed before the ISB are visible to the instructions fetched after the ISB instruction. Examples of context altering operations that might require the insertion of an ISB instruction to ensure the operations are complete are:

- ensuring a system control update has occurred
- branch predictor maintenance operations.

In addition, any branches that appear in program order after the ISB instruction are written into the branch prediction logic with the context that is visible after the ISB instruction. This is needed to ensure correct execution of the instruction stream.

Any context altering operations appearing in program order after the ISB instruction only take effect after the ISB has been executed.

An ARMv7-M implementation must choose how far ahead of the current point of execution it prefetches instructions. This can be either a fixed or a dynamically varying number of instructions. As well as choosing how many instructions to prefetch, an implementation can choose which possible future execution path to prefetch along. For example, after a branch instruction, it can prefetch either the instruction appearing in program order after 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. The memory barrier instructions, ISB, DMB or DSB as appropriate, are used to force execution ordering where necessary.

For details of the ISB instruction see *ISB* on page A6-76.

A3.8 Caches and memory hierarchy

Support for caches in ARMv7-M is limited to memory attributes. These can be exported on a supporting bus protocol such as AMBA (AHB or AXI protocols) to support system caches.

In situations where a breakdown in coherency can occur, software must manage the caches using cache maintenance operations which are memory mapped and IMPLEMENTATION DEFINED.

A3.8.1 Introduction to caches

A cache is a block of high-speed memory locations containing both address information (commonly known as a TAG) and the associated data. The purpose is to increase the average speed of a memory access. Caches operate on two principles of locality:

Spatial locality an access to one location is likely to be followed by accesses from adjacent locations, for example, sequential instruction execution or usage of a data structure

Temporal locality an access to an area of memory is likely to be repeated within a short time period, for example, execution of a code loop

To minimize the quantity of control information stored, the spatial locality property is used to group several locations together under the same TAG. This logical block is commonly known as a cache line. When data is loaded into a cache, access times for subsequent loads and stores are reduced, resulting in overall performance benefits. An access to information already in a cache is known as a cache hit, and other accesses are called cache misses.

Normally, caches are self-managing, with the updates occurring automatically. Whenever the processor wants to access a cacheable location, the cache is checked. If the access is a cache hit, the access occurs immediately, otherwise a location is allocated and the cache line loaded from memory. Different cache topologies and access policies are possible, however they must comply with the memory coherency model of the underlying architecture.

Caches introduce a number of potential problems, mainly because of:

- memory accesses occurring at times other than when the programmer would normally expect them
- the existence of multiple physical locations where a data item can be held.

A3.8.2 Implication of caches to the application programmer

Caches are largely invisible to the application programmer, but can become visible due to a breakdown in coherency. Such a breakdown can occur when:

- memory locations are updated by other agents in the systems
- memory updates made from the application code must be made visible to other agents in the system.

For example:

In systems with a DMA that reads memory locations which are held in the data cache of a processor, a breakdown of coherency occurs when the processor has written new data in the data cache, but the DMA reads the old data held in memory.

In a Harvard architecture of caches, a breakdown of coherency occurs when new instruction data has been written into the data cache and/or to memory, but the instruction cache still contains the old instruction data.

A3.8.3 Preloading caches

The ARM architecture provides memory system hints `PLD` (Preload Data) and `PLI` (Preload instruction) to permit software to communicate the expected use of memory locations to the hardware. The memory system can respond by taking actions that are expected to speed up the memory accesses if and when they do occur. The effect of these memory system hints is IMPLEMENTATION DEFINED. Typically, implementations will use this information to bring the data or instruction locations into caches that have faster access times than Normal memory.

The Preload instructions are hints, and so implementations can treat them as NOPs without affecting the functional behavior of the device. The instructions do not generate exceptions, but the memory system operations might generate an imprecise fault (asynchronous exception) due to the memory access.

Chapter A4
The ARMv7-M Instruction Set

This chapter describes the Thumb instruction set as it applies to ARMv7-M. It contains the following sections:

A4.1 About the instruction set

ARMv7-M supports a large number of 32-bit instructions that were introduced as Thumb-2 technology into the Thumb instruction set. Much of the functionality available is identical to the ARM instruction set supported alongside the Thumb instruction set in ARMv6T2 and other ARMv7 profiles. This chapter describes the functionality available in the ARMv7-M Thumb instruction set, and the *Unified Assembler Language* (UAL) that can be assembled to either the Thumb or ARM instruction sets.

Thumb instructions are either 16-bit or 32-bit, and are aligned on a two-byte boundary. 16-bit and 32-bit instructions can be intermixed freely. Many common operations are most efficiently executed using 16-bit instructions. However:

- Most 16-bit instructions can only access eight of the general purpose registers, R0-R7. These are known as the low registers. A small number of 16-bit instructions can access the high registers, R8-R15.
- Many operations that would require two or more 16-bit instructions can be more efficiently executed with a single 32-bit instruction.

The ARM and Thumb instruction sets are designed to *interwork* freely. Because ARMv7-M only supports Thumb instructions, interworking instructions in ARMv7-M must only reference Thumb state execution, see *ARMv7-M and interworking support* for more details.

In addition, see:

- Chapter A5 *Thumb Instruction Set Encoding* for encoding details of the Thumb instruction set
- Chapter A6 *Thumb Instruction Details* for detailed descriptions of the instructions.

A4.1.1 ARMv7-M and interworking support

Thumb interworking is held as bit [0] of an *interworking address*. Interworking addresses are used in the following instructions: `BX`, `BLX`, or an `LDR` or `LDM` that loads the PC.

ARMv7-M only supports the Thumb instruction execution state, therefore the value of address bit [0] must be 1 in interworking instructions, otherwise a fault occurs. All instructions ignore bit [0] and write bits [31:1]:’0’ when updating the PC.

16-bit instructions that update the PC behave as follows:

- `ADD` (register) and `MOV` (register) branch within Thumb state without interworking

  Note

  The use of Rd as the PC in the `ADD` (SP plus register) 16-bit instruction is deprecated.

- `B`, or the `B<cond>` instruction, branches without interworking
- `BLX` (register) and `BX` interwork on the value in Rm
- `POP` interworks on the value loaded to the PC
- `BKPT` and `SVC` cause exceptions and are not considered to be interworking instructions.

32-bit instructions that update the PC behave as follows:

- `B`, or the `B` instruction, branches without interworking
- `BL` branches to Thumb state based on the instruction encoding, not due to bit [0] of the value written to the PC
- `LDM` and `LDR` support interworking using the value written to the PC
- `TBB` and `TBH` branch without interworking.

For more details, see the description of the `BXWritePC()` function in *Pseudocode details of ARM core register operations* on page A2-11.

A4.1.2 Conditional execution

Conditionally executed means that the instruction only has its normal effect on the programmers' model operation, memory and coprocessors if the N, Z, C and V flags in the APSR satisfy a condition specified in the instruction. If the flags do not satisfy this condition, the instruction acts as a NOP, that is, execution advances to the next instruction as normal, including any relevant checks for exceptions being taken, but has no other effect.

Most Thumb instructions are unconditional. Conditional execution in Thumb code can be achieved using any of the following instructions:

- A 16-bit conditional branch instruction, with a branch range of –256 to +254 bytes. See *B* on page A6-40 for details. Before the additional instruction support in ARMv6T2, this was the only mechanism for conditional execution in Thumb code.
- A 32-bit conditional branch instruction, with a branch range of approximately ± 1MB. See *B* on page A6-40 for details.
- 16-bit Compare and Branch on Zero and Compare and Branch on Nonzero instructions, with a branch range of +4 to +130 bytes. See *CBNZ, CBZ* on page A6-52 for details.
- A 16-bit If-Then instruction that makes up to four following instructions conditional. See *IT* on page A6-78 for details. The instructions that are made conditional by an `IT` instruction are called its *IT block*. Instructions in an IT block must either all have the same condition, or some can have one condition, and others can have the inverse condition.

See *Conditional execution* on page A6-8 for more information about conditional execution.

A4.2 Unified Assembler Language

This document uses the ARM *Unified Assembler Language* (UAL). This assembly language syntax provides a canonical form for all ARM and Thumb instructions.

UAL describes the syntax for the mnemonic and the operands of each instruction. In addition, it assumes that instructions and data items can be given labels. It does not specify the syntax to be used for labels, nor what assembler directives and options are available. See your assembler documentation for these details.

Earlier ARM assembly language mnemonics are still supported as synonyms, as described in the instruction details.

Note

Most earlier Thumb assembly language mnemonics are *not* supported. See Appendix C *Legacy Instruction Mnemonics* for details.

UAL includes *instruction selection* rules that specify which instruction encoding is selected when more than one can provide the required functionality. For example, both 16-bit and 32-bit encodings exist for an `ADD R0,R1,R2` instruction. The most common instruction selection rule is that when both a 16-bit encoding and a 32-bit encoding are available, the 16-bit encoding is selected, to optimize code density.

Syntax options exist to override the normal instruction selection rules and ensure that a particular encoding is selected. These are useful when disassembling code, to ensure that subsequent assembly produces the original code, and in some other situations.

A4.2.1 Conditional instructions

For maximum portability of UAL assembly language between the ARM and Thumb instruction sets, ARM recommends that:

- `IT` instructions are written before conditional instructions in the correct way for the Thumb instruction set.
- When assembling to the ARM instruction set, assemblers check that any `IT` instructions are correct, but do not generate any code for them.

Although other Thumb instructions are unconditional, all instructions that are made conditional by an `IT` instruction must be written with a condition. These conditions must match the conditions imposed by the `IT` instruction. For example, an `ITTEE EQ` instruction imposes the `EQ` condition on the first two following instructions, and the `NE` condition on the next two. Those four instructions must be written with `EQ`, `EQ`, `NE` and `NE` conditions respectively.

Some instructions cannot be made conditional by an `IT` instruction. Some instructions can be conditional if they are the last instruction in the IT block, but not otherwise.

The branch instruction encodings that include a condition field cannot be made conditional by an `IT` instruction. If the assembler syntax indicates a conditional branch that correctly matches a preceding `IT` instruction, it is assembled using a branch instruction encoding that does not include a condition field.

A4.2.2 Use of labels in UAL instruction syntax

The UAL syntax for some instructions includes the label of an instruction or a literal data item that is at a fixed offset from the instruction being specified. The assembler must:

1. Calculate the `PC` or `Align(PC,4)` value of the instruction. The `PC` value of an instruction is its address plus 4 for a Thumb instruction, or plus 8 for an ARM instruction. The `Align(PC,4)` value of an instruction is its `PC` value ANDed with `0xFFFFFFFC` to force it to be word-aligned. There is no difference between the `PC` and `Align(PC,4)` values for an ARM instruction, but there can be for a Thumb instruction.

2. Calculate the offset from the `PC` or `Align(PC,4)` value of the instruction to the address of the labelled instruction or literal data item.

3. Assemble a *PC-relative* encoding of the instruction, that is, one that reads its `PC` or `Align(PC,4)` value and adds the calculated offset to form the required address.

   Note

   For instructions that encode a subtraction operation, if the instruction cannot encode the calculated offset, but can encode minus the calculated offset, the instruction encoding specifies a subtraction of minus the calculated offset.

The syntax of the following instructions includes a label:

- `B`, `BL`, and `BLX` (immediate). The assembler syntax for these instructions always specifies the label of the instruction that they branch to. Their encodings specify a sign-extended immediate offset that is added to the `PC` value of the instruction to form the target address of the branch.

- `CBNZ` and `CBZ`. The assembler syntax for these instructions always specifies the label of the instruction that they branch to. Their encodings specify a zero-extended immediate offset that is added to the `PC` value of the instruction to form the target address of the branch. They do not support backward branches.

- `LDC`, `LDC2`, `LDR`, `LDRB`, `LDRD`, `LDRH`, `LDRSB`, `LDRSH`, `PLD`, `and PLI`. The normal assembler syntax of these load instructions can specify the label of a literal data item that is to be loaded. The encodings of these instructions specify a zero-extended immediate offset that is either added to or subtracted from the `Align(PC,4)` value of the instruction to form the address of the data item. A few such encodings perform a fixed addition or a fixed subtraction and must only be used when that operation is required, but most contain a bit that specifies whether the offset is to be added or subtracted.

  When the assembler calculates an offset of 0 for the normal syntax of these instructions, it must assemble an encoding that adds 0 to the `Align(PC,4)` value of the instruction. Encodings that subtract 0 from the `Align(PC,4)` value cannot be specified by the normal syntax.

  There is an alternative syntax for these instructions that specifies the addition or subtraction and the immediate offset explicitly. In this syntax, the label is replaced by `[PC, #+/-<imm>]`, where:

  `+/-` Is `+` or omitted to specify that the immediate offset is to be added to the `Align(PC,4)` value, or `-` if it is to be subtracted.

  `<imm>` Is the immediate offset.

This alternative syntax makes it possible to assemble the encodings that subtract 0 from the `Align(PC,4)` value, and to disassemble them to a syntax that can be re-assembled correctly.

- `ADR`. The normal assembler syntax for this instruction can specify the label of an instruction or literal data item whose address is to be calculated. Its encoding specifies a zero-extended immediate offset that is either added to or subtracted from the `Align(PC,4)` value of the instruction to form the address of the data item, and some opcode bits that determine whether it is an addition or subtraction.

  When the assembler calculates an offset of 0 for the normal syntax of this instruction, it must assemble the encoding that adds 0 to the `Align(PC,4)` value of the instruction. The encoding that subtracts 0 from the `Align(PC,4)` value cannot be specified by the normal syntax.

  There is an alternative syntax for this instruction that specifies the addition or subtraction and the immediate value explicitly, by writing them as additions `ADD <Rd>,PC,#<imm>` or subtractions `SUB <Rd>,PC,#<imm>`. This alternative syntax makes it possible to assemble the encoding that subtracts 0 from the `Align(PC,4)` value, and to disassemble it to a syntax that can be re-assembled correctly.

Note

ARM recommends that where possible, you avoid using:

- the alternative syntax for the `ADR`, `LDC`, `LDC2`, `LDR`, `LDRB`, `LDRD`, `LDRH`, `LDRSB`, `LDRSH`, `PLD`, `and PLI` instructions
- the encodings of these instructions that subtract 0 from the `Align(PC,4)` value.

A4.3 Branch instructions

Table A4-1 summarizes the branch instructions in the Thumb instruction set. In addition to providing for changes in the flow of execution, some branch instructions can change instruction set.

Table A4-1 Branch instructions

Instruction	Usage	Range
B on page A6-40	Branch to target address	+/–1 MB
CBNZ, *CBZ* on page A6-52	Compare and Branch on Nonzero, Compare and Branch on Zero	0-126 B
BL on page A6-49	Call a subroutine	+/–16 MB
BLX (register) on page A6-50	Call a subroutine, optionally change instruction set	Any
BX on page A6-51	Branch to target address, change instruction set	Any
TBB, *TBH* on page A6-258	Table Branch (byte offsets)	0-510 B
	Table Branch (halfword offsets)	0-131070 B

`LDR` and `LDM` instructions can also cause a branch. See *Load and store instructions* on page A4-16 and *Load/store multiple instructions* on page A4-19 for details.

A4.4 Data-processing instructions

Core data-processing instructions belong to one of the following groups:

- *Standard data-processing instructions*. This group perform basic data-processing operations, and share a common format with some variations.
- *Shift instructions* on page A4-10.
- *Multiply instructions* on page A4-11.
- *Saturating instructions* on page A4-12.
- *Packing and unpacking instructions* on page A4-13.
- *Miscellaneous data-processing instructions* on page A4-14.
- *Divide instructions* on page A4-14.

A4.4.1 Standard data-processing instructions

These instructions generally have a destination register Rd, a first operand register Rn, and a second operand. The second operand can be either another register Rm, or a modified immediate constant.

If the second operand is a modified immediate constant, it is encoded in 12 bits of the instruction. See *Modified immediate constants in Thumb instructions* on page A5-15 for details.

If the second operand is another register, it can optionally be shifted in any of the following ways:

`LSL`	Logical Shift Left by 1-31 bits.
`LSR`	Logical Shift Right by 1-32 bits.
`ASR`	Arithmetic Shift Right by 1-32 bits.
`ROR`	Rotate Right by 1-31 bits.
`RRX`	Rotate Right with Extend. See *Shift and rotate operations* on page A2-5 for details.

In Thumb code, the amount to shift by is always a constant encoded in the instruction.

In addition to placing a result in the destination register, these instructions can optionally set the condition code flags, according to the result of the operation. If they do not set the flags, existing flag settings from a previous instruction are preserved.

Table A4-2 on page A4-9 summarizes the main data-processing instructions in the Thumb instruction set. Generally, each of these instructions is described in two sections in Chapter A6 *Thumb Instruction Details*, one section for each of the following:

- `INSTRUCTION` (immediate) where the second operand is a modified immediate constant.
- `INSTRUCTION` (register) where the second operand is a register, or a register shifted by a constant.

Table A4-2 Standard data-processing instructions

Mnemonic	Instruction	Notes
ADC	Add with Carry	-
ADD	Add	Thumb permits use of a modified immediate constant or a zero-extended 12-bit immediate constant.
ADR	Form PC-relative Address	First operand is the PC. Second operand is an immediate constant. Thumb supports a zero-extended 12-bit immediate constant. Operation is an addition or a subtraction.
AND	Bitwise AND	-
BIC	Bitwise Bit Clear	-
CMN	Compare Negative	Sets flags. Like ADD but with no destination register.
CMP	Compare	Sets flags. Like SUB but with no destination register.
EOR	Bitwise Exclusive OR	-
MOV	Copies operand to destination	Has only one operand, with the same options as the second operand in most of these instructions. If the operand is a shifted register, the instruction is an LSL, LSR, ASR, or ROR instruction instead. See *Shift instructions* on page A4-10 for details. Thumb permits use of a modified immediate constant or a zero-extended 16-bit immediate constant.
MVN	Bitwise NOT	Has only one operand, with the same options as the second operand in most of these instructions.
ORN	Bitwise OR NOT	-
ORR	Bitwise OR	-
RSB	Reverse Subtract	Subtracts first operand from second operand. This permits subtraction from constants and shifted registers.
SBC	Subtract with Carry	-
SUB	Subtract	Thumb permits use of a modified immediate constant or a zero-extended 12-bit immediate constant.
TEQ	Test Equivalence	Sets flags. Like EOR but with no destination register.
TST	Test	Sets flags. Like AND but with no destination register.

A4.4.2 Shift instructions

Table A4-3 lists the shift instructions in the Thumb instruction set.

Table A4-3 Shift instructions

Instruction	See
Arithmetic Shift Right	*ASR (immediate)* on page A6-36
Arithmetic Shift Right	*ASR (register)* on page A6-38
Logical Shift Left	*LSL (immediate)* on page A6-134
Logical Shift Left	*LSL (register)* on page A6-136
Logical Shift Right	*LSR (immediate)* on page A6-138
Logical Shift Right	*LSR (register)* on page A6-140
Rotate Right	*ROR (immediate)* on page A6-194
Rotate Right	*ROR (register)* on page A6-196
Rotate Right with Extend	*RRX* on page A6-198

A4.4.3 Multiply instructions

These instructions can operate on signed or unsigned quantities. In some types of operation, the results are same whether the operands are signed or unsigned.

- Table A4-4 summarizes the multiply instructions where there is no distinction between signed and unsigned quantities.

  The least significant 32 bits of the result are used. More significant bits are discarded.

- Table A4-5 summarizes the signed multiply instructions.
- Table A4-6 summarizes the unsigned multiply instructions.

Table A4-4 General multiply instructions

Instruction	Operation (number of bits)
MLA on page A6-146	32 = 32 + 32 x 32
MLS on page A6-147	32 = 32 – 32 x 32
MUL on page A6-160	32 = 32 x 32

Table A4-5 Signed multiply instructions

Instruction	Operation (number of bits)
SMLAL on page A6-213	64 = 64 + 32 x 32
SMULL on page A6-214	64 = 32 x 32

Table A4-6 Unsigned multiply instructions

Instruction	Operation (number of bits)
UMLAL on page A6-268	64 = 64 + 32 x 32
UMULL on page A6-269	64 = 32 x 32

A4.4.4 Saturating instructions

Table A4-7 lists the saturating instructions in the Thumb instruction set. See *Pseudocode details of saturation* on page A2-9 for more information.

Table A4-7 Core saturating instructions

Instruction	See	Operation
Signed Saturate	*SSAT* on page A6-215	Saturates optionally shifted 32-bit value to selected range
Unsigned Saturate	*USAT* on page A6-270	Saturates optionally shifted 32-bit value to selected range

A4.4.5 Packing and unpacking instructions

Table A4-8 lists the packing and upacking instructions in the Thumb instruction set.

Table A4-8 Packing and unpacking instructions

Instruction	See	Operation
Signed Extend Byte	*SXTB* on page A6-254	Extend 8 bits to 32
Signed Extend Halfword	*SXTH* on page A6-256	Extend 16 bits to 32
Unsigned Extend Byte	*UXTB* on page A6-272	Extend 8 bits to 32
Unsigned Extend Halfword	*UXTH* on page A6-274	Extend 16 bits to 32

A4.4.6 Miscellaneous data-processing instructions

Table A4-9 lists the miscellaneous data-processing instructions in the Thumb instruction set. Immediate values in these instructions are simple binary numbers.

Table A4-9 Miscellaneous data-processing instructions

Instruction	See	Notes
Bit Field Clear	*BFC* on page A6-42	-
Bit Field Insert	*BFI* on page A6-43	-
Count Leading Zeros	*CLZ* on page A6-57	-
Move Top	*MOVT* on page A6-153	Moves 16-bit immediate value to top halfword. Bottom halfword unaltered.
Reverse Bits	*RBIT* on page A6-190	-
Byte-Reverse Word	*REV* on page A6-191	-
Byte-Reverse Packed Halfword	*REV16* on page A6-192	-
Byte-Reverse Signed Halfword	*REVSH* on page A6-193	-
Signed Bit Field Extract	*SBFX* on page A6-208	-
Unsigned Bit Field Extract	*UBFX* on page A6-266	-

A4.4.7 Divide instructions

In the ARMv7-M profile, the Thumb instruction set includes signed and unsigned integer divide instructions that are implemented in hardware. For details of the instructions see:

- *SDIV* on page A6-210
- *UDIV* on page A6-267.

In the ARMv7-M profile, the CCR.DIV_0_TRP bit enables divide by zero fault detection:

DZ == 0 Divide-by-zero returns a zero result.

DZ == 1 `SDIV` and `UDIV` generate an Undefined Instruction exception on a divide-by-zero.

The CCR.DIV_0_TRP bit is cleared to zero on reset.

A4.5 Status register access instructions

The `MRS` and `MSR` instructions move the contents of the *Application Program Status Register* (APSR) to or from a general-purpose register.

The APSR is described in *The Application Program Status Register (APSR)* on page A2-13.

The condition flags in the APSR are normally set by executing data-processing instructions, and are normally used to control the execution of conditional instructions. However, you can set the flags explicitly using the `MSR` instruction, and you can read the current state of the flags explicitly using the `MRS` instruction.

For details of the system level use of status register access instructions `CPS`, `MRS` and `MSR`, see Chapter B4 *ARMv7-M System Instructions*.

A4.6 Load and store instructions

Table A4-10 summarizes the general-purpose register load and store instructions in the Thumb instruction set. See also *Load/store multiple instructions* on page A4-19.

Load and store instructions have several options for addressing memory. See *Addressing modes* on page A4-18 for more information.

Table A4-10 Load and store instructions

Data type	Load	Store	Load unprivileged	Store unprivileged	Load exclusive	Store exclusive
32-bit word	LDR	STR	LDRT	STRT	LDREX	STREX
16-bit halfword	-	STRH	-	STRHT	-	STREXH
16-bit unsigned halfword	LDRH	-	LDRHT	-	LDREXH	-
16-bit signed halfword	LDRSH	-	LDRSHT	-	-	-
8-bit byte	-	STRB	-	STRBT	-	STREXB
8-bit unsigned byte	LDRB	-	LDRBT	-	LDREXB	-
8-bit signed byte	LDRSB	-	LDRSBT	-	-	-
two 32-bit words	LDRD	STRD	-	-	-	-

A4.6.1 Loads to the PC

The `LDR` instruction can be used to load a value into the PC. The value loaded is treated as an interworking address, as described by the `LoadWritePC()` pseudocode function in *Pseudocode details of ARM core register operations* on page A2-11.

A4.6.2 Halfword and byte loads and stores

Halfword and byte stores store the least significant halfword or byte from the register, to 16 or 8 bits of memory respectively. There is no distinction between signed and unsigned stores.

Halfword and byte loads load 16 or 8 bits from memory into the least significant halfword or byte of a register. Unsigned loads zero-extend the loaded value to 32 bits, and signed loads sign-extend the value to 32 bits.

A4.6.3 Unprivileged loads and stores

In an unprivileged mode, unprivileged loads and stores operate in exactly the same way as the corresponding ordinary operations. In a privileged mode, unprivileged loads and stores are treated as though they were executed in an unprivileged mode. See *Privilege level access controls for data accesses* on page A3-28 for more information.

A4.6.4 Exclusive loads and stores

Exclusive loads and stores provide for shared memory synchronization. See *Synchronization and semaphores* on page A3-8 for more information.

A4.6.5 Addressing modes

The address for a load or store is formed from two parts: a value from a base register, and an offset.

The base register can be any one of the general-purpose registers.

For loads, the base register can be the PC. This permits PC-relative addressing for position-independent code. Instructions marked (literal) in their title in Chapter A6 *Thumb Instruction Details* are PC-relative loads.

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 value. 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.

Register The offset is a value from a general-purpose register. This register cannot be the PC. The value can be added to, or subtracted from, the base register value. Register offsets are useful for accessing arrays or blocks of data.

Scaled register The offset is a general-purpose register, other than the PC, shifted by an immediate value, then added to or subtracted from the base register. This means an array index can be scaled by the size of each array element.

The offset and base register can be used in three different ways to form the memory address. The addressing modes are described as follows:

Offset The offset is added to or subtracted from the base register to form the memory address.

Pre-indexed The offset is added to or subtracted from the base register to form the memory address. The base register is then updated with this new address, to permit automatic indexing through an array or memory block.

Post-indexed The value of the base register alone is used as the memory address. The offset is then added to or subtracted from the base register. and this value is stored back in the base register, to permit automatic indexing through an array or memory block.

Note

Not every variant is available for every instruction, and the range of permitted immediate values and the options for scaled registers vary from instruction to instruction. See Chapter A6 *Thumb Instruction Details* for full details for each instruction.

A4.7 Load/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.

The memory locations are consecutive word-aligned words. The addresses used are obtained from a base register, and can be either above or below the value in the base register. The base register can optionally be updated by the total size of the data transferred.

Table A4-11 summarizes the load/store multiple instructions in the Thumb instruction set.

Table A4-11 Load/store multiple instructions

Instruction	Description
Load Multiple, Increment After or Full Descending	*LDM / LDMIA / LDMFD* on page A6-84
Load Multiple, Decrement Before or Empty Ascending	*LDMDB / LDMEA* on page A6-86
Pop multiple registers off the stack [a]	*POP* on page A6-186
Push multiple registers onto the stack [b]	*PUSH* on page A6-188
Store Multiple, Increment After or Empty Ascending	*STM / STMIA / STMEA* on page A6-218
Store Multiple, Decrement Before or Full Descending	*STMDB / STMFD* on page A6-220

[a] This instruction is equivalent to an `LDM` instruction with the SP as base register, and base register updating.

[b] This instruction is equivalent to an `STMDB` instruction with the SP as base register, and base register updating.

A4.7.1 Loads to the PC

The `LDM`, `LDMDB`, and `POP` instructions can be used to load a value into the PC. The value loaded is treated as an interworking address, as described by the `LoadWritePC()` pseudocode function in *Pseudocode details of ARM core register operations* on page A2-11.

A4.8 Miscellaneous instructions

Table A4-12 summarizes the miscellaneous instructions in the Thumb instruction set.

Table A4-12 Miscellaneous instructions

Instruction	See
Clear Exclusive	*CLREX* on page A6-56
Debug hint	*DBG* on page A6-67
Data Memory Barrier	*DMB* on page A6-68
Data Synchronization Barrier	*DSB* on page A6-70
Instruction Synchronization Barrier	*ISB* on page A6-76
If Then (makes following instructions conditional)	*IT* on page A6-78
No Operation	*NOP* on page A6-167
Preload Data	*PLD*~~*, PLDW*~~ *(immediate)* on page A6-176
	PLD (register) on page A6-180
Preload Instruction	*PLI (immediate, literal)* on page A6-182
	PLI (register) on page A6-184
Send Event	*SEV* on page A6-212
Supervisor Call	*SVC (formerly SWI)* on page A6-252
Wait for Event	*WFE* on page A6-276
Wait for Interrupt	*WFI* on page A6-277
Yield	*YIELD* on page A6-278

A4.9 Exception-generating instructions

The following instructions are intended specifically to cause a processor exception to occur:

- The Supervisor Call (`SVC`, formerly `SWI`) instruction is used to cause an SVC exception to occur. This is the main mechanism for unprivileged (User) code to make calls to privileged Operating System code. See *Exception model* on page B1-14 for details.
- The Breakpoint (`BKPT`) instruction provides for software breakpoints. It can generate a debug monitor exception or cause a running system to halt depending on the debug configuration. See *Debug event behavior* on page C1-14 for more details.

A4.10 Coprocessor instructions

There are three types of instruction for communicating with coprocessors. These permit the processor to:

- Initiate a coprocessor data-processing operation. See *CDP, CDP2* on page A6-54 for details.
- Transfer general-purpose registers to and from coprocessor registers. For details, see:
  - *MCR, MCR2* on page A6-142
  - *MCRR, MCRR2* on page A6-144
  - *MRC, MRC2* on page A6-154
  - *MRRC, MRRC2* on page A6-156.
- Generate addresses for the coprocessor load/store instructions. For details, see:
  - *LDC, LDC2 (immediate)* on page A6-80
  - *LDC, LDC2 (literal)* on page A6-82
  - *STC, STC2* on page A6-216.

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 the core processor, ignoring non-coprocessor instructions and coprocessor instructions for other coprocessors. Coprocessor instructions that cannot be executed by any coprocessor hardware generate a UsageFault exception and record the reason as follows:

- Where access is denied to a coprocessor by the Coprocessor Access Register, the UFSR.NOCP flag is set to indicate the coprocessor does not exist.
- Where the coprocessor access is allowed but the instruction is unknown, the UFSR.UNDEFINSTR flag is set to indicate that the instruction is UNDEFINED.

Chapter A5
Thumb Instruction Set Encoding

This chapter introduces the Thumb instruction set and describes how it uses the ARM programmers' model. It contains the following sections:

A5.1 Thumb instruction set encoding

The Thumb instruction stream is a sequence of halfword-aligned halfwords. Each Thumb instruction is either a single 16-bit halfword in that stream, or a 32-bit instruction consisting of two consecutive halfwords in that stream.

If bits [15:11] of the halfword being decoded take any of the following values, the halfword is the first halfword of a 32-bit instruction:

- 0b11101
- 0b11110
- 0b11111.

Otherwise, the halfword is a 16-bit instruction.

See *16-bit Thumb instruction encoding* on page A5-5 for details of the encoding of 16-bit Thumb instructions.

See *32-bit Thumb instruction encoding* on page A5-13 for details of the encoding of 32-bit Thumb instructions.

A5.1.1 UNDEFINED and UNPREDICTABLE instruction set space

An attempt to execute an unallocated instruction results in either:

- Unpredictable behavior. The instruction is described as UNPREDICTABLE.
- An Undefined Instruction exception. The instruction is described as UNDEFINED.

An instruction is UNDEFINED if it is declared as UNDEFINED in an instruction description, or in this chapter

An instruction is UNPREDICTABLE if:

- a bit marked (0) or (1) in the encoding diagram of an instruction is not 0 or 1 respectively, and the pseudocode for that encoding does not indicate that a different special case applies
- it is declared as UNPREDICTABLE in an instruction description or in this chapter.

Unless otherwise specified:

- Thumb instructions introduced in an architecture variant are either UNPREDICTABLE or UNDEFINED in earlier architecture variants.
- A Thumb instruction that is provided by one or more of the architecture extensions is either UNPREDICTABLE or UNDEFINED in an implementation that does not include those extensions.

In both cases, the instruction is UNPREDICTABLE if it is a 32-bit instruction in an architecture variant before ARMv6T2, and UNDEFINED otherwise.

A5.1.2 Use of 0b1111 as a register specifier

The use of 0b1111 as a register specifier is not normally permitted in Thumb instructions. When a value of 0b1111 is permitted, a variety of meanings is possible. For register reads, these meanings are:

- Read the PC value, that is, the address of the current instruction + 4. The base register of the table branch instructions TBB and TBH can be the PC. This enables branch tables to be placed in memory immediately after the instruction. (Some instructions read the PC value implicitly, without the use of a register specifier, for example the conditional branch instruction B<cond>.)

  Note

  Use of the PC as the base register in the STC instruction is deprecated in ARMv7.

- Read the word-aligned PC value, that is, the address of the current instruction + 4, with bits [1:0] forced to zero. The base register of LDC, LDR, LDRB, LDRD (pre-indexed, no writeback), LDRH, LDRSB, and LDRSH instructions can be the word-aligned PC. This enables PC-relative data addressing. In addition, some encodings of the ADD and SUB instructions permit their source registers to be 0b1111 for the same purpose.

- Read zero. This is done in some cases when one instruction is a special case of another, more general instruction, but with one operand zero. In these cases, the instructions are listed on separate pages, with a special case in the pseudocode for the more general instruction cross-referencing the other page.

For register writes, these meanings are:

- The PC can be specified as the destination register of an LDR instruction. This is done by encoding Rt as 0b1111. The loaded value is treated as an address, and the effect of execution is a branch to that address. bit [0] of the loaded value selects the execution state after the branch and must have the value 1.

  Some other instructions write the PC in similar ways, either implicitly (for example, B<cond>) or by using a register mask rather than a register specifier (LDM). The address to branch to can be a loaded value (for example, LDM), a register value (for example, BX), or the result of a calculation (for example, TBB or TBH).

- Discard the result of a calculation. This is done in some cases when one instruction is a special case of another, more general instruction, but with the result discarded. In these cases, the instructions are listed on separate pages, with a special case in the pseudocode for the more general instruction cross-referencing the other page.

- If the destination register specifier of an LDRB, LDRH, LDRSB, or LDRSH instruction is 0b1111, the instruction is a memory hint instead of a load operation.

- If the destination register specifier of an MRC instruction is 0b1111, bits [31:28] of the value transferred from the coprocessor are written to the N, Z, C, and V flags in the APSR, and bits [27:0] are discarded.

A5.1.3 Use of 0b1101 as a register specifier

R13 is defined in the Thumb instruction set so that its use is primarily as a stack pointer, and R13 is normally identified as SP in Thumb instructions. In 32-bit Thumb instructions, if you use R13 as a general purpose register beyond the architecturally defined constraints described in this section, the results are UNPREDICTABLE.

The restrictions applicable to R13 are described in:

- *R13[1:0] definition*
- *32-bit Thumb instruction support for R13*.

See also *16-bit Thumb instruction support for R13*.

R13[1:0] definition

Bits [1:0] of R13 are treated as *SBZP* (Should Be Zero or Preserved). Writing a non-zero value to bits [1:0] results in UNPREDICTABLE behavior. Reading bits [1:0] returns zero.

32-bit Thumb instruction support for R13

R13 instruction support is restricted to the following:

- R13 as the source or destination register of a `MOV` instruction. Only register to register transfers without shifts are supported, with no flag setting:

```
MOV     SP,Rm
MOV     Rn,SP
```

- Adjusting R13 up or down by a multiple of its alignment:

```
ADD{W}  SP,SP,#N           ; For N a multiple of 4
SUB{W}  SP,SP,#N           ; For N a multiple of 4
ADD     SP,SP,Rm,LSL #shft  ; For shft=0,1,2,3
SUB     SP,SP,Rm,LSL #shft  ; For shft=0,1,2,3
```

- R13 as a base register (Rn) of any load or store instruction. This supports SP-based addressing for load, store, or memory hint instructions, with positive or negative offsets, with and without writeback.

- R13 as the first operand (Rn) in any `ADD{S}`, `CMN`, `CMP`, or `SUB{S}` instruction. The add and subtract instructions support SP-based address generation, with the address going into a general-purpose register. `CMN` and `CMP` are useful for stack checking in some circumstances.

- R13 as the transferred register (Rt) in any `LDR` or `STR` instruction.

- R13 as the address in a `POP` or `PUSH` instruction.

16-bit Thumb instruction support for R13

For 16-bit data processing instructions that affect high registers, R13 can only be used as described in *32-bit Thumb instruction support for R13*. Any other use is deprecated. This affects the high register forms of `CMP` and `ADD`, where the use of R13 as Rm is deprecated.

A5.2 16-bit Thumb instruction encoding

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
opcode															

Table A5-1 shows the allocation of 16-bit instruction encodings.

Table A5-1 16-bit Thumb instruction encoding

opcode	Instruction or instruction class
00xxxx	*Shift (immediate), add, subtract, move, and compare* on page A5-6
010000	*Data processing* on page A5-7
010001	*Special data instructions and branch and exchange* on page A5-8
01001x	Load from Literal Pool, see *LDR (literal)* on page A6-90
0101xx 011xxx 100xxx	*Load/store single data item* on page A5-9
10100x	Generate PC-relative address, see *ADR* on page A6-30
10101x	Generate SP-relative address, see *ADD (SP plus immediate)* on page A6-26
1011xx	*Miscellaneous 16-bit instructions* on page A5-10
11000x	Store multiple registers, see *STM / STMIA / STMEA* on page A6-218
11001x	Load multiple registers, see *LDM / LDMIA / LDMFD* on page A6-84
1101xx	*Conditional branch, and supervisor call* on page A5-12
11100x	Unconditional Branch, see *B* on page A6-40

A5.2.1 Shift (immediate), add, subtract, move, and compare

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	opcode													

Table A5-2 shows the allocation of encodings in this space.

Table A5-2 16-bit shift(immediate), add, subtract, move and compare encoding

opcode	Instruction	See
000xx	Logical Shift Left	*LSL (immediate)* on page A6-134
001xx	Logical Shift Right	*LSR (immediate)* on page A6-138
010xx	Arithmetic Shift Right	*ASR (immediate)* on page A6-36
01100	Add register	*ADD (register)* on page A6-24
01101	Subtract register	*SUB (register)* on page A6-246
01110	Add 3-bit immediate	*ADD (immediate)* on page A6-22
01111	Subtract 3-bit immediate	*SUB (immediate)* on page A6-244
100xx	Move	*MOV (immediate)* on page A6-148
101xx	Compare	*CMP (immediate)* on page A6-62
110xx	Add 8-bit immediate	*ADD (immediate)* on page A6-22
111xx	Subtract 8-bit immediate	*SUB (immediate)* on page A6-244

A5.2.2 Data processing

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	0	opcode									

Table A5-3 shows the allocation of encodings in this space.

Table A5-3 16-bit data processing instructions

opcode	Instruction	See
0000	Bitwise AND	*AND (register)* on page A6-34
0001	Exclusive OR	*EOR (register)* on page A6-74
0010	Logical Shift Left	*LSL (register)* on page A6-136
0011	Logical Shift Right	*LSR (register)* on page A6-140
0100	Arithmetic Shift Right	*ASR (register)* on page A6-38
0101	Add with Carry	*ADC (register)* on page A6-20
0110	Subtract with Carry	*SBC (register)* on page A6-206
0111	Rotate Right	*ROR (register)* on page A6-196
1000	Set flags on bitwise AND	*TST (register)* on page A6-264
1001	Reverse Subtract from 0	*RSB (immediate)* on page A6-200
1010	Compare Registers	*CMP (register)* on page A6-64
1011	Compare Negative	*CMN (register)* on page A6-60
1100	Logical OR	*ORR (register)* on page A6-174
1101	Multiply Two Registers	*MUL* on page A6-160
1110	Bit Clear	*BIC (register)* on page A6-46
1111	Bitwise NOT	*MVN (register)* on page A6-164

A5.2.3 Special data instructions and branch and exchange

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	1	opcode									

Table A5-4 shows the allocation of encodings in this space.

Table A5-4 Special data instructions and branch and exchange

opcode	Instruction	See
00xx	Add Registers	*ADD (register)* on page A6-24
0100	UNPREDICTABLE	
0101 011x	Compare Registers	*CMP (register)* on page A6-64
10xx	Move Registers	*MOV (register)* on page A6-150
110x	Branch and Exchange	*BX* on page A6-51
111x	Branch with Link and Exchange	*BLX (register)* on page A6-50

A5.2.4 Load/store single data item

15 14 13 12	11 10 9	8 7 6 5 4 3 2 1 0
opA	opB	

These instructions have one of the following values in opA:

- 0b0101
- 0b011x
- 0b100x.

Table A5-5 shows the allocation of encodings in this space.

Table A5-5 16-bit Load/store instructions

opA	opB	Instruction	See
0101	000	Store Register	*STR (register)* on page A6-224
0101	001	Store Register Halfword	*STRH (register)* on page A6-240
0101	010	Store Register Byte	*STRB (register)* on page A6-228
0101	011	Load Register Signed Byte	*LDRSB (register)* on page A6-122
0101	100	Load Register	*LDR (register)* on page A6-92
0101	101	Load Register Halfword	*LDRH (register)* on page A6-114
0101	110	Load Register Byte	*LDRB (register)* on page A6-98
0101	111	Load Register Signed Halfword	*LDRSH (register)* on page A6-130
0110	0xx	Store Register	*STR (immediate)* on page A6-222
0110	1xx	Load Register	*LDR (immediate)* on page A6-88
0111	0xx	Store Register Byte	*STRB (immediate)* on page A6-226
0111	1xx	Load Register Byte	*LDRB (immediate)* on page A6-94
1000	0xx	Store Register Halfword	*STRH (immediate)* on page A6-238
1000	1xx	Load Register Halfword	*LDRH (immediate)* on page A6-110
1001	0xx	Store Register SP relative	*STR (immediate)* on page A6-222
1001	1xx	Load Register SP relative	*LDR (immediate)* on page A6-88

A5.2.5 Miscellaneous 16-bit instructions

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	opcode											

Table A5-6 shows the allocation of encodings in this space. Other encodings in this space are UNDEFINED.

Table A5-6 Miscellaneous 16-bit instructions

opcode	Instruction	See
0110011	Change Processor State	*CPS* on page B4-2
00000xx	Add Immediate to SP	*ADD (SP plus immediate)* on page A6-26
00001xx	Subtract Immediate from SP	*SUB (SP minus immediate)* on page A6-248
0001xxx	Compare and Branch on Zero	*CBNZ, CBZ* on page A6-52
001000x	Signed Extend Halfword	*SXTH* on page A6-256
001001x	Signed Extend Byte	*SXTB* on page A6-254
001010x	Unsigned Extend Halfword	*UXTH* on page A6-274
001011x	Unsigned Extend Byte	*UXTB* on page A6-272
0011xxx	Compare and Branch on Zero	*CBNZ, CBZ* on page A6-52
010xxxx	Push Multiple Registers	*PUSH* on page A6-188
1001xxx	Compare and Branch on Nonzero	*CBNZ, CBZ* on page A6-52
101000x	Byte-Reverse Word	*REV* on page A6-191
101001x	Byte-Reverse Packed Halfword	*REV16* on page A6-192
101011x	Byte-Reverse Signed Halfword	*REVSH* on page A6-193
1011xxx	Compare and Branch on Nonzero	*CBNZ, CBZ* on page A6-52
110xxxx	Pop Multiple Registers	*POP* on page A6-186
1110xxx	Breakpoint	*BKPT* on page A6-48
1111xxx	If-Then, and hints	*If-Then, and hints* on page A5-11

If-Then, and hints

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	1	1	1	opA				opB			

Table A5-7 shows the allocation of encodings in this space.

Other encodings in this space are unallocated hints. They execute as NOPs, but software must not use them.

Table A5-7 If-Then and hint instructions

opA	opB	Instruction	See
xxxx	not 0000	If-Then	*IT* on page A6-78
0000	0000	No Operation hint	*NOP* on page A6-167
0001	0000	Yield hint	*YIELD* on page A6-278
0010	0000	Wait for Event hint	*WFE* on page A6-276
0011	0000	Wait for Interrupt hint	*WFI* on page A6-277
0100	0000	Send Event hint	*SEV* on page A6-212

A5.2.6 Conditional branch, and supervisor call

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	opcode											

Table A5-8 shows the allocation of encodings in this space.

Table A5-8 Branch and supervisor call instructions

opcode	Instruction	See
not 111x	Conditional branch	*B* on page A6-40
1110	Permanently UNDEFINED	
1111	Supervisor call	*SVC (formerly SWI)* on page A6-252

A5.3 32-bit Thumb instruction encoding

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	op1		op2											op															

op1 != 0b00. If op1 == 0b00, a 16-bit instruction is encoded, see *16-bit Thumb instruction encoding* on page A5-5.

Table A5-9 shows the allocation of ARMv7-M Thumb encodings in this space.

Table A5-9 32-bit Thumb encoding

op1	op2	op	Instruction class
01	00xx 0xx	x	*Load/store multiple* on page A5-20
01	00xx 1xx	x	*Load/store dual or exclusive, table branch* on page A5-21
01	01xx xxx	x	*Data processing (shifted register)* on page A5-26
01	1xxx xxx	x	*Coprocessor instructions* on page A5-32
10	x0xx xxx	0	*Data processing (modified immediate)* on page A5-14
10	x1xx xxx	0	*Data processing (plain binary immediate)* on page A5-17
10	xxxx xxx	1	*Branches and miscellaneous control* on page A5-18
11	000x xx0	x	*Store single data item* on page A5-25
11	00xx 001	x	*Load byte, memory hints* on page A5-24
11	00xx 011	x	*Load halfword, unallocated memory hints* on page A5-23
11	00xx 101	x	*Load word* on page A5-22
11	00xx 111	x	UNDEFINED
11	010x xxx	x	*Data processing (register)* on page A5-28
11	0110 xxx	x	*Multiply, and multiply accumulate* on page A5-30
11	0111 xxx	x	*Long multiply, long multiply accumulate, and divide* on page A5-31
11	1xxx xxx	x	*Coprocessor instructions* on page A5-32

A5.3.1 Data processing (modified immediate)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0		0	op					Rn				0				Rd											

Table A5-10 shows the allocation of encodings in this space. Other encodings in this space are UNDEFINED.

Table A5-10 32-bit modified immediate data processing instructions

op	Rn	Rd	Instruction	See
0000x		not 1111	Bitwise AND	*AND (immediate)* on page A6-32
		1111	Test	*TST (immediate)* on page A6-262
0001x			Bitwise Clear	*BIC (immediate)* on page A6-44
0010x	not 1111		Bitwise Inclusive OR	*ORR (immediate)* on page A6-172
	1111		Move	*MOV (immediate)* on page A6-148
0011x	not 1111		Bitwise OR NOT	*ORN (immediate)* on page A6-168
	1111		Bitwise NOT	*MVN (immediate)* on page A6-162
0100x		not 1111	Bitwise Exclusive OR	*EOR (immediate)* on page A6-72
		1111	Test Equivalence	*TEQ (immediate)* on page A6-260
1000x		not 1111	Add	*ADD (immediate)* on page A6-22
		1111	Compare Negative	*CMN (immediate)* on page A6-58
1010x			Add with Carry	*ADC (immediate)* on page A6-18
1011x			Subtract with Carry	*SBC (immediate)* on page A6-204
1101x		not 1111	Subtract	*SUB (immediate)* on page A6-244
		1111	Compare	*CMP (immediate)* on page A6-62
1110x			Reverse Subtract	*RSB (immediate)* on page A6-200

These instructions all have modified immediate constants, rather than a simple 12-bit binary number. This provides a more useful range of values. See *Modified immediate constants in Thumb instructions* on page A5-15 for details.

A5.3.2 Modified immediate constants in Thumb instructions

<table>
<tr><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td colspan="5"></td><td>i</td><td colspan="10"></td><td></td><td colspan="3">imm3</td><td colspan="4"></td><td>a</td><td>b</td><td>c</td><td>d</td><td>e</td><td>f</td><td>g</td><td>h</td></tr>
</table>

Table A5-11 shows the range of modified immediate constants available in Thumb data processing instructions, and how they are encoded in the a, b, c, d, e, f, g, h, i, and imm3 fields in the instruction.

Table A5-11 Encoding of modified immediates in Thumb data-processing instructions

i:imm3:a	<const> [a]
0000x	`00000000 00000000 00000000 abcdefgh`
0001x	`00000000 abcdefgh 00000000 abcdefgh` [b]
0010x	`abcdefgh 00000000 abcdefgh 00000000` [b]
0011x	`abcdefgh abcdefgh abcdefgh abcdefgh` [b]
01000	`1bcdefgh 00000000 00000000 00000000`
01001	`01bcdefg h0000000 00000000 00000000`
01010	`001bcdef gh000000 00000000 00000000`
01011	`0001bcde fgh00000 00000000 00000000`
. . .	. . 8-bit values shifted to other positions .
11101	`00000000 00000000 000001bc defgh000`
11110	`00000000 00000000 0000001b cdefgh00`
11111	`00000000 00000000 00000001 bcdefgh0`

a. In this table, the immediate constant value is shown in binary form, to relate `abcdefgh` to the encoding diagram. In assembly syntax, the immediate value is specified in the usual way (a decimal number by default).

b. UNPREDICTABLE if abcdefgh == 00000000.

Carry out

A logical operation with i:imm3:a == ’00xxx’ does not affect the carry flag. Otherwise, a logical operation that sets the flags sets the Carry flag to the value of bit [31] of the modified immediate constant.

Operation

```
// ThumbExpandImm()
// ================

bits(32) ThumbExpandImm(bits(12) imm12)

    // APSR.C argument to following function call does not affect the imm32 result.
    (imm32, -) = ThumbExpandImm_C(imm12, APSR.C);

    return imm32;

// ThumbExpandImm_C()
// ==================

(bits(32), bit) ThumbExpandImm_C(bits(12) imm12, bit carry_in)

    if imm12<11:10> == '00' then

        case imm12<9:8> of
            when '00'
                imm32 = ZeroExtend(imm12<7:0>, 32);
            when '01'
                if imm12<7:0> == '00000000' then UNPREDICTABLE;
                imm32 = '00000000' : imm12<7:0> : '00000000' : imm12<7:0>;
            when '10'
                if imm12<7:0> == '00000000' then UNPREDICTABLE;
                imm32 = imm12<7:0> : '00000000' : imm12<7:0> : '00000000';
            when '11'
                if imm12<7:0> == '00000000' then UNPREDICTABLE;
                imm32 = imm12<7:0> : imm12<7:0> : imm12<7:0> : imm12<7:0>;
        carry_out = carry_in;

    else

        unrotated_value = ZeroExtend('1':imm12<6:0>, 32);
        (imm32, carry_out) = ROR_C(unrotated_value, UInt(imm12<11:7>));

    return (imm32, carry_out);
```

A5.3.3 Data processing (plain binary immediate)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0		1	op					Rn				0															

Table A5-10 on page A5-14 shows the allocation of encodings in this space. Other encodings in this space are UNDEFINED.

Table A5-12 32-bit unmodified immediate data processing instructions

op	Rn	Instruction	See
00000	not 1111	Add Wide (12-bit)	*ADD (immediate)* on page A6-22
	1111	Form PC-relative Address	*ADR* on page A6-30
00100		Move Wide (16-bit)	*MOV (immediate)* on page A6-148
01010	not 1111	Subtract Wide (12-bit)	*SUB (immediate)* on page A6-244
	1111	Form PC-relative Address	*ADR* on page A6-30
01100		Move Top (16-bit)	*MOVT* on page A6-153
~~100x0~~ [a]		Signed Saturate	*SSAT* on page A6-215
10100		Signed Bit Field Extract	*SBFX* on page A6-208
10110	not 1111	Bit Field Insert	*BFI* on page A6-43
	1111	Bit Field Clear	*BFC* on page A6-42
~~110x0~~ [a]		Unsigned Saturate	*USAT* on page A6-270
11100		Unsigned Bit Field Extract	*UBFX* on page A6-266

[a] In the second halfword of the instruction, bits [14:12.7:6] != 0b00000.

A5.3.4 Branches and miscellaneous control

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	op1											1	op2														

Table A5-13 shows the allocation of encodings in this space. Other encodings in this space are UNDEFINED.

Table A5-13 Branches and miscellaneous control instructions

op2	op1	Instruction	See
0x0	not x111xxx	Conditional branch	*B* on page A6-40
0x0	011100x	Move to Special Register	*MSR (register)* on page A6-159
0x0	0111010	-	*Hint instructions* on page A5-19
0x0	0111011	-	*Miscellaneous control instructions* on page A5-19
0x0	011111x	Move from Special Register	*MRS* on page A6-158
010	1111111	Permanently UNDEFINED	-
0x1	xxxxxxx	Branch	*B* on page A6-40
~~1x0~~	~~xxxxxxx~~		
1x1	xxxxxxx	Branch with Link	*BL* on page A6-49

Hint instructions

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	0	1	0					1	0		0		op1			op2							

Table A5-14 shows the allocation of encodings in this space. Other encodings in this space are unallocated hints that execute as NOPs. These unallocated hint encodings are reserved and software must not use them.

Table A5-14 Change Processor State, and hint instructions

op1	op2	Instruction	See
not 000	xxxx xxxx	UNDEFINED[a]	
000	0000 0000	No Operation hint	*NOP* on page A6-167
000	0000 0001	Yield hint	*YIELD* on page A6-278
000	0000 0010	Wait For Event hint	*WFE* on page A6-276
000	0000 0011	Wait For Interrupt hint	*WFI* on page A6-277
000	0000 0100	Send Event hint	*SEV* on page A6-212
000	1111 xxxx	Debug hint	*DBG* on page A6-67

[a] These encodings provide a 32-bit form of the `CPS` instruction in the ARMv7-A and ARMv7-R architecture profiles.

Miscellaneous control instructions

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	0	1	1					1	0		0					op							

Table A5-15 shows the allocation of encodings in this space. Other encodings in this space are UNDEFINED in ARMv7-M.

Table A5-15 Miscellaneous control instructions

op	Instruction	See
0010	Clear Exclusive	*CLREX* on page A6-56
0100	Data Synchronization Barrier	*DSB* on page A6-70
0101	Data Memory Barrier	*DMB* on page A6-68
0110	Instruction Synchronization Barrier	*ISB* on page A6-76

A5.3.5 Load/store multiple

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	0	0	op		0	W	L	Rn																			

Table A5-16 shows the allocation of encodings in this space. Other encodings in this space are UNDEFINED.

Table A5-16 Load/store multiple instructions

op	L	W:Rn	Instruction	See
01	0		Store Multiple (Increment After, Empty Ascending)	*STM / STMIA / STMEA* on page A6-218
01	1	not 11101	Load Multiple (Increment After, Full Descending)	*LDM / LDMIA / LDMFD* on page A6-84
01	1	11101	Pop Multiple Registers from the stack	*POP* on page A6-186
10	0	not 11101	Store Multiple (Decrement Before, Full Descending)	*STMDB / STMFD* on page A6-220
10	0	11101	Push Multiple Registers to the stack.	*PUSH* on page A6-188
10	1		Load Multiple (Decrement Before, Empty Ascending)	*LDMDB / LDMEA* on page A6-86

A5.3.6 Load/store dual or exclusive, table branch

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	0	0	op1		1	op2		Rn												op3							

Table A5-17 shows the allocation of encodings in this space. Other encodings in this space are UNDEFINED.

Table A5-17 Load/store dual or exclusive, table branch

op1	op2	op3	Instruction	See
00	00	xxxx	Store Register Exclusive	*STREX* on page A6-234
00	01	xxxx	Load Register Exclusive	*LDREX* on page A6-106
0x 1x	10 x0	xxxx xxxx	Store Register Dual	*STRD (immediate)* on page A6-232
0x 1x	11 x1	xxxx xxxx	Load Register Dual	*LDRD (immediate)* on page A6-102, *LDRD (literal)* on page A6-104
01	00	0100	Store Register Exclusive Byte	*STREXB* on page A6-235
01	00	0101	Store Register Exclusive Halfword	*STREXH* on page A6-236
01	01	0000	Table Branch Byte	*TBB, TBH* on page A6-258
01	01	0001	Table Branch Halfword	*TBB, TBH* on page A6-258
01	01	0100	Load Register Exclusive Byte	*LDREXB* on page A6-107
01	01	0101	Load Register Exclusive Halfword	*LDREXH* on page A6-108

A5.3.7 Load word

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	op1		1	0	1	Rn								op2											

Table A5-18 shows the allocation of encodings in this space. Other encodings in this space are UNDEFINED.

Table A5-18 Load word

op1	op2	Rn	Instruction	See
01	xxxxxx	not 1111	Load Register	*LDR (immediate)* on page A6-88
00	1xx1xx	not 1111		
00	1100xx	not 1111		
00	1110xx	not 1111	Load Register Unprivileged	*LDRT* on page A6-133
00	000000	not 1111	Load Register	*LDR (register)* on page A6-92
0x	xxxxxx	1111	Load Register	*LDR (literal)* on page A6-90

A5.3.8 Load halfword, unallocated memory hints

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	op1		0	1	1	Rn				Rt				op2											

Table A5-19 shows the allocation of encodings in this space. Other encodings in this space are UNDEFINED.

Table A5-19 Load halfword

op1	op2	Rn	Rt	Instruction	See
01	xxxxxx	not 1111	not 1111	Load Register Halfword	*LDRH (immediate)* on page A6-110
00	1xx1xx	not 1111	not 1111		
00	1100xx	not 1111	not 1111		
00	1110xx	not 1111	not 1111	Load Register Halfword Unprivileged	*LDRHT* on page A6-116
0x	xxxxxx	1111	not 1111	Load Register Halfword	*LDRH (literal)* on page A6-112
00	000000	not 1111	not 1111	Load Register Halfword	*LDRH (register)* on page A6-114
11	xxxxxx	not 1111	not 1111	Load Register Signed Halfword	*LDRSH (immediate)* on page A6-126
10	1xx1xx	not 1111	not 1111		
10	1100xx	not 1111	not 1111		
10	1110xx	not 1111	not 1111	Load Register Signed Halfword Unprivileged	*LDRSHT* on page A6-132
1x	xxxxxx	1111	not 1111	Load Register Signed Halfword	*LDRSH (literal)* on page A6-128
10	000000	not 1111	not 1111	Load Register Signed Halfword	*LDRSH (register)* on page A6-130
~~xx~~	~~xxxxxx~~	~~xxxxxx~~	~~1111~~	~~Unallocated memory hint~~ [a]	-

a. Unallocated memory hints must be implemented as `NOP`, and software must not use them.

A5.3.9 Load byte, memory hints

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	op1		0	0	1	Rn				Rt				op2											

Table A5-20 shows the allocation of encodings in this space. Other encodings in this space are UNDEFINED.

Table A5-20 Load byte, preload

op1	op2	Rn	Rt	Instruction	See
01	xxxxxx	not 1111	not 1111	Load Register Byte	*LDRB (immediate)* on page A6-94
00	1xx1xx	not 1111			
00	1100xx	not 1111	not 1111		
00	1110xx	not 1111		Load Register Byte Unprivileged	*LDRBT* on page A6-100
0x	xxxxxx	1111	not 1111	Load Register Byte	*LDRB (literal)* on page A6-96
00	000000	not 1111	not 1111	Load Register Byte	*LDRB (register)* on page A6-98
11	xxxxxx	not 1111	not 1111	Load Register Signed Byte	*LDRSB (immediate)* on page A6-118
10	1xx1xx	not 1111			
10	1100xx	not 1111	not 1111		
10	1110xx	not 1111		Load Register Signed Byte Unprivileged	*LDRSBT* on page A6-124
1x	xxxxxx	1111	not 1111	Load Register Signed Byte	*LDRSB (literal)* on page A6-120
10	000000	not 1111	not 1111	Load Register Signed Byte	*LDRSB (register)* on page A6-122
01	xxxxxx	not 1111	1111	Preload Data	*PLD~~, PLDW~~ (immediate)* on page A6-176
00	1100xx	not 1111	1111		
0x	xxxxxx	1111	1111		
00	000000	not 1111	1111	Preload Data	*PLD (register)* on page A6-180
11	xxxxxx	not 1111	1111	Preload Instruction	*PLI (immediate, literal)* on page A6-182
10	1100xx	not 1111	1111		
1x	xxxxxx	1111	1111		
10	000000	not 1111	1111	Preload Instruction	*PLI (register)* on page A6-184

A5.3.10 Store single data item

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	0	op1			0									op2											

Table A5-21 show the allocation of encodings in this space. Other encodings in this space are UNDEFINED.

Table A5-21 Store single data item

op1	op2	Instruction	See
100	xxxxxx	Store Register Byte	*STRB (immediate)* on page A6-226
000	1xxxxx		
000	0xxxxx	Store Register Byte	*STRB (register)* on page A6-228
101	xxxxxx	Store Register Halfword	*STRH (immediate)* on page A6-238
001	1xxxxx		
001	0xxxxx	Store Register Halfword	*STRH (register)* on page A6-240
110	xxxxxx	Store Register	*STR (immediate)* on page A6-222
010	1xxxxx		
010	0xxxxx	Store Register	*STR (register)* on page A6-224

A5.3.11 Data processing (shifted register)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	0	1	op				S	Rn								Rd											

Table A5-22 shows the allocation of encodings in this space.

Other encodings in this space are UNDEFINED.

Table A5-22 Data-processing (shifted register)

op	Rn	Rd	S	Instruction	See
0000	-	not 1111	x	Bitwise AND	*AND (register)* on page A6-34
		1111	0	UNPREDICTABLE	-
			1	Test	*TST (register)* on page A6-264
0001	-	-	-	Bitwise Bit Clear	*BIC (register)* on page A6-46
0010	not 1111	-	-	Bitwise OR	*ORR (register)* on page A6-174
	1111	-	-	-	*Move register and immediate shifts* on page A5-27
0011	not 1111	-	-	Bitwise OR NOT	*ORN (register)* on page A6-170
	1111	-	-	Bitwise NOT	*MVN (register)* on page A6-164
0100	-	not 1111	-	Bitwise Exclusive OR	*EOR (register)* on page A6-74
		1111	0	UNPREDICTABLE	-
			1	Test Equivalence	*TEQ (register)* on page A6-261
1000	-	not 1111	-	Add	*ADD (register)* on page A6-24
		1111	0	UNPREDICTABLE	-
			1	Compare Negative	*CMN (register)* on page A6-60
1010	-	-	-	Add with Carry	*ADC (register)* on page A6-20
1011	-	-	-	Subtract with Carry	*SBC (register)* on page A6-206

Table A5-22 Data-processing (shifted register) (continued)

op	Rn	Rd	S	Instruction	See
1101	-	not 1111	-	Subtract	*SUB (register)* on page A6-246
		1111	0	UNPREDICTABLE	-
			1	Compare	*CMP (register)* on page A6-64
1110	-	-	-	Reverse Subtract	*RSB (register)* on page A6-202

Move register and immediate shifts

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	0	1	0	0	1	0		1	1	1	1		imm3							imm2		type					

Table A5-23 shows the allocation of encodings in this space.

Table A5-23 Move register and immediate shifts

type	imm3:imm2	Instruction	See
00	00000	Move	*MOV (register)* on page A6-150
	not 00000	Logical Shift Left	*LSL (immediate)* on page A6-134
01	-	Logical Shift Right	*LSR (immediate)* on page A6-138
10	-	Arithmetic Shift Right	*ASR (immediate)* on page A6-36
11	00000	Rotate Right with Extend	*RRX* on page A6-198
	not 00000	Rotate Right	*ROR (immediate)* on page A6-194

A5.3.12 Data processing (register)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	1	0	op1				Rn				1	1	1	1					op2							

If, in the second halfword of the instruction, bits [15:12] != 0b1111, the instruction is UNDEFINED.

Table A5-24 shows the allocation of encodings in this space. Other encodings in this space are UNDEFINED.

Table A5-24 Data processing (register)

op1	op2	Instruction	See
000x	0000	Logical Shift Left	*LSL (register)* on page A6-136
001x	0000	Logical Shift Right	*LSR (register)* on page A6-140
010x	0000	Arithmetic Shift Right	*ASR (register)* on page A6-38
011x	0000	Rotate Right	*ROR (register)* on page A6-196
0000	1xxx	Signed Extend Halfword	*SXTH* on page A6-256[a]
0001	1xxx	Unsigned Extend Halfword	*UXTH* on page A6-274[a]
0100	1xxx	Signed Extend Byte	*SXTB* on page A6-254[a]
0101	1xxx	Unsigned Extend Byte	~~*UXTH* on page A6-274~~[a]
10xx	10xx	See *Miscellaneous operations* on page A5-29	

[a] where Rn == ’1111’

A5.3.13 Miscellaneous operations

<table>
<tr><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>0</td><td>1</td><td>0</td><td>1</td><td>0</td><td colspan="2">op1</td><td colspan="4"></td><td>1</td><td>1</td><td>1</td><td>1</td><td colspan="4"></td><td>1</td><td>0</td><td colspan="2">op2</td><td colspan="4"></td></tr>
</table>

If, in the second halfword of the instruction, bits [15:12] != 0b1111, the instruction is UNDEFINED.

Table A5-25 shows the allocation of encodings in this space. Other encodings in this space are UNDEFINED.

Table A5-25 Miscellaneous operations

op1	op2	Instruction	See
01	00	Byte-Reverse Word	*REV* on page A6-191
01	01	Byte-Reverse Packed Halfword	*REV16* on page A6-192
01	10	Reverse Bits	*RBIT* on page A6-190
01	11	Byte-Reverse Signed Halfword	*REVSH* on page A6-193
11	00	Count Leading Zeros	*CLZ* on page A6-57

A5.3.14 Multiply, and multiply accumulate

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	1	1	0	op1							Ra								0	0	op2					

If, in the second halfword of the instruction, bits [7:6] != 0b00, the instruction is UNDEFINED.

Table A5-26 shows the allocation of encodings in this space. Other encodings in this space are UNDEFINED.

Table A5-26 Multiply, and multiply accumulate operations

op1	op2	Ra	Instruction	See
000	00	not 1111	Multiply Accumulate	*MLA* on page A6-146
000	00	1111	Multiply	*MUL* on page A6-160
000	01		Multiply and Subtract	*MLS* on page A6-147

A5.3.15 Long multiply, long multiply accumulate, and divide

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	1	1	1	op1															op2							

Table A5-27 shows the allocation of encodings in this space. Other encodings in this space are UNDEFINED.

Table A5-27 Long multiply, long multiply accumulate, and divide operations

op1	op2	Instruction	See
000	0000	Signed Multiply Long	*SMULL* on page A6-214
001	1111	Signed Divide	*SDIV* on page A6-210
010	0000	Unsigned Multiply Long	*UMULL* on page A6-269
011	1111	Unsigned Divide	*UDIV* on page A6-267
100	0000	Signed Multiply Accumulate Long	*SMLAL* on page A6-213
110	0000	Unsigned Multiply Accumulate Long	*UMLAL* on page A6-268

A5.3.16 Coprocessor instructions

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1		1	1	op1														coproc							op				

Table A5-28 shows the allocation of encodings in this space. Other encodings in this space ~~and where the target coprocessor does not exist~~ are UNDEFINED.

Table A5-28 Coprocessor instructions

op1	op	coproc	Instructions	See
0xxxx0[a]	x	xxxx	Store Coprocessor	*STC, STC2* on page A6-216
0xxxx1[a]	x	xxxx	Load Coprocessor	*LDC, LDC2 (immediate)* on page A6-80, *LDC, LDC2 (literal)* on page A6-82
000100	x	xxxx	Move to Coprocessor from two ARM core registers	*MCRR, MCRR2* on page A6-144
000101	x	xxxx	Move to two ARM core registers from Coprocessor	*MRRC, MRRC2* on page A6-156
10xxxx	0	xxxx	Coprocessor data operations	*CDP, CDP2* on page A6-54
10xxx0	1	xxxx	Move to Coprocessor from ARM core register	*MCR, MCR2* on page A6-142
10xxx1	1	xxxx	Move to ARM core register from Coprocessor	*MRC, MRC2* on page A6-154

[a] but not 000x0x

Chapter A6
Thumb Instruction Details

This chapter describes Thumb[®] instruction support in ARMv7-M. It contains the following sections:

- *Format of instruction descriptions* on page A6-2
- *Standard assembler syntax fields* on page A6-7
- *Conditional execution* on page A6-8
- *Shifts applied to a register* on page A6-12
- *Memory accesses* on page A6-15
- *Hint Instructions* on page A6-16.
- *Alphabetical list of ARMv7-M Thumb instructions* on page A6-17.

A6.1 Format of instruction descriptions

The instruction descriptions in the alphabetical lists of instructions in *Alphabetical list of ARMv7-M Thumb instructions* on page A6-17 normally use the following format:

- instruction section title
- introduction to the instruction
- instruction encoding(s) with architecture information
- assembler syntax
- pseudocode describing how the instruction operates
- exception information
- notes (where applicable).

Each of these items is described in more detail in the following subsections.

A few instruction descriptions describe alternative mnemonics for other instructions and use an abbreviated and modified version of this format.

A6.1.1 Instruction section title

The instruction section title gives the base mnemonic for the instructions described in the section. When one mnemonic has multiple forms described in separate instruction sections, this is followed by a short description of the form in parentheses. The most common use of this is to distinguish between forms of an instruction in which one of the operands is an immediate value and forms in which it is a register.

Parenthesized text is also used to document the former mnemonic in some cases where a mnemonic has been replaced entirely by another mnemonic in the new assembler syntax.

A6.1.2 Introduction to the instruction

The instruction section title is followed by text that briefly describes the main features of the instruction. This description is not necessarily complete and is not definitive. If there is any conflict between it and the more detailed information that follows, the latter takes priority.

A6.1.3 Instruction encodings

The *Encodings* subsection contains a list of one or more instruction encodings. For reference purposes, each Thumb instruction encoding is labelled, T1, T2, T3...

Each instruction encoding description consists of:

- Information about which architecture variants include the particular encoding of the instruction. Thumb instructions present since ARMv4T are labelled as *all versions of the Thumb ISA*, otherwise:
  - ARMv5T* means all variants of ARM Architecture version 5 that include Thumb instruction support.
  - ARMv6-M means a Thumb-only variant of the ARM architecture microcontroller profile that is compatible with ARMv6 Thumb support prior to the introduction of Thumb-2 technology.
  - ARMv7-M means a Thumb-only variant of the ARM architecture microcontroller profile that provides enhanced performance and functionality with respect to ARMv6-M through Thumb-2 technology and additional system features such as fault handling support.

  Note

  This manual does not provide architecture variant information about non-M profile variants of ARMv6 and ARMv7. For such information, see the *ARM Architecture Reference Manual*.

- An assembly syntax that ensures that the assembler selects the encoding in preference to any other encoding. In some cases, multiple syntaxes are given. The correct one to use is sometimes indicated by annotations to the syntax, such as *Inside IT block* and *Outside IT block*. In other cases, the correct one to use can be determined by looking at the assembler syntax description and using it to determine which syntax corresponds to the instruction being disassembled.

  There is usually more than one syntax that ensures re-assembly to any particular encoding, and the exact set of syntaxes that do so usually depends on the register numbers, immediate constants and other operands to the instruction. For example, when assembling to the Thumb instruction set, the syntax `AND R0,R0,R8` ensures selection of a 32-bit encoding but `AND R0,R0,R1` selects a 16-bit encoding.

  The assembly syntax documented for the encoding is chosen to be the simplest one that ensures selection of that encoding for all operand combinations supported by that encoding. This often means that it includes elements that are only necessary for a small subset of operand combinations. For example, the assembler syntax documented for the 32-bit Thumb `AND` (register) encoding includes the `.W` qualifier to ensure that the 32-bit encoding is selected even for the small proportion of operand combinations for which the 16-bit encoding is also available.

  The assembly syntax given for an encoding is therefore a suitable one for a disassembler to disassemble that encoding to. However, disassemblers may wish to use simpler syntaxes when they are suitable for the operand combination, in order to produce more readable disassembled code.

- An encoding diagram. This is half-width for 16-bit Thumb encodings and full-width for 32-bit Thumb encodings. The 32-bit Thumb encodings use a double vertical line between the two halfwords to act as a reminder that 32-bit Thumb encodings use the byte order of a sequence of two halfwords rather than of a word, as described in *Instruction alignment and byte ordering* on page A3-6.

- Encoding-specific pseudocode. This is pseudocode that translates the encoding-specific instruction fields into inputs to the encoding-independent pseudocode in the later *Operation* subsection, and that picks out any special cases in the encoding. For a detailed description of the pseudocode used and of the relationship between the encoding diagram, the encoding-specific pseudocode and the encoding-independent pseudocode, see Appendix G *Pseudocode definition*.

A6.1.4 Assembler syntax

The *Assembly syntax* subsection describes the standard UAL syntax for the instruction.

Each syntax description consists of the following elements:

- One or more syntax prototype lines written in a typewriter font, using the conventions described in *Assembler syntax prototype line conventions* on page A6-5. Each prototype line documents the mnemonic and (where appropriate) operand parts of a full line of assembler code. When there is more than one such line, each prototype line is annotated to indicate required results of the encoding-specific pseudocode. For each instruction encoding, this information can be used to determine whether any instructions matching that encoding are available when assembling that syntax, and if so, which ones.

- The line *where:* followed by descriptions of all of the variable or optional fields of the prototype syntax line.

  Some syntax fields are standardized across all or most instructions. These fields are described in *Standard assembler syntax fields* on page A6-7.

  By default, syntax fields that specify registers (such as `<Rd>`, `<Rn>`, or `<Rt>`) are permitted to be any of R0-R12 or LR in Thumb instructions. These require that the encoding-specific pseudocode should set the corresponding integer variable (such as `d`, `n`, or `t`) to the corresponding register number (0-12 for R0-R12, 14 for LR). This can normally be done by setting the corresponding bitfield in the instruction (named Rd, Rn, Rt...) to the binary encoding of that number. In the case of 16-bit Thumb encodings, this bitfield is normally of length 3 and so the encoding is only available when one of R0-R7 was specified in the assembler syntax. It is also common for such encodings to use a bitfield name such as Rdn. This indicates that the encoding is only available if `<Rd>` and `<Rn>` specify the same register, and that the register number of that register is encoded in the bitfield if they do.

  The description of a syntax field that specifies a register sometimes extends or restricts the permitted range of registers or document other differences from the default rules for such fields. Typical extensions are to allow the use of the SP and/or the PC (using register numbers 13 and 15 respectively).

Note

The pre-UAL Thumb assembler syntax is incompatible with UAL and is not documented in the instruction sections.

Assembler syntax prototype line conventions

The following conventions are used in assembler syntax prototype lines and their subfields:

`< >` 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 a Thumb 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. This is often done by specifying a required output from the encoding-specific pseudocode, such as add = `TRUE`. The assembler must only use encodings that produce that output.

`{ }` 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.

Many instructions have an optional destination register. Unless otherwise stated, if such a destination register is omitted, it is the same as the immediately following source register in the instruction syntax.

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.

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 description of the variable item indicates how they must be used.

A6.1.5 Pseudocode describing how the instruction operates

The *Operation* subsection contains encoding-independent pseudocode that describes the main operation of the instruction. For a detailed description of the pseudocode used and of the relationship between the encoding diagram, the encoding-specific pseudocode and the encoding-independent pseudocode, see Appendix G *Pseudocode definition*.

A6.1.6 Exception information

The *Exceptions* subsection contains a list of the exceptional conditions that can be caused by execution of the instruction.

Processor exceptions are listed as follows:

- Resets and interrupts (including NMI, PendSV and SysTick) are not listed. They can occur before or after the execution of any instruction, and in some cases during the execution of an instruction, but they are not in general caused by the instruction concerned.
- MemManage and BusFault exceptions are listed for all instructions that perform explicit data memory accesses.

  All instruction fetches can cause MemManage and BusFault exceptions. These are not caused by execution of the instruction and so are not listed.
- UsageFault exceptions can occur for a variety of reasons and are listed against instructions as appropriate.

  UsageFault exceptions also occur when pseudocode indicates that the instruction is UNDEFINED. These UsageFaults are not listed.
- The SVCall exception is listed for the `SVC` instruction.
- The DebugMonitor exception is listed for the `BKPT` instruction.
- HardFault exceptions can arise from escalation of faults listed against an instruction, but are not themselves listed.

Note

For a summary of the different types of MemManage, BusFault and UsageFault exceptions see *Fault behavior* on page B1-39.

A6.1.7 Notes

Where appropriate, additional notes about the instruction appear under further subheadings.

A6.2 Standard assembler syntax fields

The following assembler syntax fields are standard across all or most instructions:

`<c>` Is an optional field. It specifies the condition under which the instruction is executed. If `<c>` is omitted, it defaults to *always* (`AL`). For details see *Conditional execution* on page A4-3.

`<q>` Specifies optional assembler qualifiers on the instruction. The following qualifiers are defined:

- `.N` Meaning narrow, specifies that the assembler must select a 16-bit encoding for the instruction. If this is not possible, an assembler error is produced.
- `.W` Meaning wide, specifies that the assembler must select a 32-bit encoding for the instruction. If this is not possible, an assembler error is produced.

If neither `.W` nor `.N` is specified, the assembler can select either 16-bit or 32-bit encodings. If both are available, it must select a 16-bit encoding. In a few cases, more than one encoding of the same length can be available for an instruction. The rules for selecting between such encodings are instruction-specific and are part of the instruction description.

A6.3 Conditional execution

Most Thumb instructions in ARMv7-M can be executed conditionally, based on the values of the APSR condition flags. The available conditions are listed in Table A6-1.

In Thumb instructions, the condition (if it is not AL) is normally encoded in a preceding IT instruction, see *Conditional instructions* on page A4-4, *ITSTATE* on page A6-10 and *IT* on page A6-78 for details. Some conditional branch instructions do not require a preceding IT instruction, and include a condition code in their encoding.

Table A6-1 Condition codes

cond	Mnemonic extension	Meaning (integer)	Meaning (floating-point) [a][b]	Condition flags
0000	EQ	Equal	Equal	Z == 1
0001	NE	Not equal	Not equal, or unordered	Z == 0
0010	CS [c]	Carry set	Greater than, equal, or unordered	C == 1
0011	CC [d]	Carry clear	Less than	C == 0
0100	MI	Minus, negative	Less than	N == 1
0101	PL	Plus, positive or zero	Greater than, equal, or unordered	N == 0
0110	VS	Overflow	Unordered	V == 1
0111	VC	No overflow	Not unordered	V == 0
1000	HI	Unsigned higher	Greater than, or unordered	C == 1 and Z == 0
1001	LS	Unsigned lower or same	Less than or equal	C == 0 or Z == 1
1010	GE	Signed greater than or equal	Greater than or equal	N == V
1011	LT	Signed less than	Less than, or unordered	N != V
1100	GT	Signed greater than	Greater than	Z == 0 and N == V
1101	LE	Signed less than or equal	Less than, equal, or unordered	Z == 1 or N != V
1110	None (AL) [e]	Always (unconditional)	Always (unconditional)	Any

[a] Unordered means at least one NaN operand.

[b] ARMv7-M does not currently support floating point instructions. This column can be ignored.

[c] HS (unsigned higher or same) is a synonym for CS.

[d] LO (unsigned lower) is a synonym for CC.

[e] AL is an optional mnemonic extension for always, except in IT instructions. See *IT* on page A6-78 for details.

A6.3.1 Pseudocode details of conditional execution

The `CurrentCond()` pseudocode function has prototype:

```
bits(4) CurrentCond()
```

and returns a 4-bit condition specifier as follows:

- For the T1 and T3 encodings of the Branch instruction (see *B* on page A6-40), it returns the 4-bit 'cond' field of the encoding.
- For all other Thumb instructions, it returns ITSTATE.IT[7:4]. See *ITSTATE* on page A6-10.

The `ConditionPassed()` function uses this condition specifier and the APSR condition flags to determine whether the instruction must be executed:

```
// ConditionPassed()
// =================

boolean ConditionPassed()
    cond = CurrentCond();

    // Evaluate base condition.
    case cond<3:1> of
        when '000'  result = (APSR.Z == '1');                          // EQ or NE
        when '001'  result = (APSR.C == '1');                          // CS or CC
        when '010'  result = (APSR.N == '1');                          // MI or PL
        when '011'  result = (APSR.V == '1');                          // VS or VC
        when '100'  result = (APSR.C == '1') && (APSR.Z == '0');     // HI or LS
        when '101'  result = (APSR.N == APSR.V);                       // GE or LT
        when '110'  result = (APSR.N == APSR.V) && (APSR.Z == '0');  // GT or LE
        when '111'  result = TRUE;                                     // AL

    // Condition bits in the set "111x" indicate the instruction is always executed.
    // Otherwise, invert condition if necessary.
    if cond<0> == '1' && cond != '1111' then
        result = !result;

    return result;
```

A6.3.2 Conditional execution of undefined instructions

If an UNDEFINED instruction fails a condition check in ARMv7-M, the instruction behaves as a NOP and does not cause an exception.

Note

The Branch (`B`) instruction with a conditional field of ‘1110’ is UNDEFINED and takes an exception unless qualified by a condition check failure from an `IT` instruction.

A6.3.3 ITSTATE

7	6	5	4	3	2	1	0
IT[7:0]							

This field holds the If-Then execution state bits for the Thumb `IT` instruction. See *IT* on page A6-78 for a description of the `IT` instruction and the associated IT block.

ITSTATE divides into two subfields:

IT[7:5] Holds the *base condition* for the current IT block. The base condition is the top 3 bits of the condition specified by the IT instruction.

This subfield is 0b000 when no IT block is active.

IT[4:0] Encodes:

- The size of the IT block. This is the number of instructions that are to be conditionally executed. The size of the block is implied by the position of the least significant 1 in this field, as shown in Table A6-2 on page A6-11.
- The value of the least significant bit of the condition code for each instruction in the block.

Note

Changing the value of the least significant bit of a condition code from 0 to 1 has the effect of inverting the condition code.

This subfield is 0b00000 when no IT block is active.

When an IT instruction is executed, these bits are set according to the condition in the instruction, and the *Then* and *Else* (T and E) parameters in the instruction, see *IT* on page A6-78 for more information.

An instruction in an IT block is conditional, see *Conditional instructions* on page A4-4. The condition used is the current value of IT[7:4]. When an instruction in an IT block completes its execution normally, `ITSTATE` is advanced to the next line of Table A6-2 on page A6-11.

See *Exception entry behavior* on page B1-21 for details of what happens if such an instruction takes an exception.

Note

Instructions that can complete their normal execution by branching are only permitted in an IT block as its last instruction, and so always result in `ITSTATE` advancing to normal execution.

Table A6-2 Effect of IT execution state bits

	IT bits [a]					
[7:5]	**[4]**	**[3]**	**[2]**	**[1]**	**[0]**	
cond_base	P1	P2	P3	P4	1	Entry point for 4-instruction IT block
cond_base	P1	P2	P3	1	0	Entry point for 3-instruction IT block
cond_base	P1	P2	1	0	0	Entry point for 2-instruction IT block
cond_base	P1	1	0	0	0	Entry point for 1-instruction IT block
000	0	0	0	0	0	Normal execution, not in an IT block

[a] Combinations of the IT bits not shown in this table are reserved.

Pseudocode details of ITSTATE operation

ITSTATE advances after normal execution of an IT block instruction. This is described by the ITAdvance() pseudocode function:

```
// ITAdvance()
// ===========

ITAdvance()
    if ITSTATE<2:0> == '000' then
        ITSTATE.IT = '00000000';
    else
        ITSTATE.IT<4:0> = LSL(ITSTATE.IT<4:0>, 1);
```

The following functions test whether the current instruction is in an IT block, and whether it is the last instruction of an IT block:

```
// InITBlock()
// ===========

boolean InITBlock()
    return (ITSTATE.IT<3:0> != '0000');

// LastInITBlock()
// ===============

boolean LastInITBlock()
    return (ITSTATE.IT<3:0> == '1000');
```

A6.4 Shifts applied to a register

ARM register offset load/store word and unsigned byte instructions can apply a wide range of different constant shifts to the offset register. Both Thumb and ARM data-processing instructions can apply the same range of different constant shifts to the second operand register. See *Constant shifts* for details.

ARM data-processing instructions can apply a register-controlled shift to the second operand register.

A6.4.1 Constant shifts

These are the same in Thumb and ARM instructions, except that the input bits come from different positions.

`<shift>` is an optional shift to be applied to `<Rm>`. It can be any one of:

(omitted) Equivalent to `LSL #0`.

`LSL #<n>` logical shift left `<n>` bits. 0 <= `<n>` <= 31.

`LSR #<n>` logical shift right `<n>` bits. 1 <= `<n>` <= 32.

`ASR #<n>` arithmetic shift right `<n>` bits. 1 <= `<n>` <= 32.

`ROR #<n>` rotate right `<n>` bits. 1 <= `<n>` <= 31.

`RRX` rotate right one bit, with extend. bit [0] is written to `shifter_carry_out`, bits [31:1] are shifted right one bit, and the Carry Flag is shifted into bit [31].

Encoding

The assembler encodes `<shift>` into two type bits and five immediate bits, as follows:

(omitted) type = 0b00, immediate = 0.

`LSL #<n>` type = 0b00, immediate = `<n>`.

`LSR #<n>` type = 0b01.
If `<n>` < 32, immediate = `<n>`.
If `<n>` == 32, immediate = 0.

`ASR #<n>` type = 0b10.
If `<n>` < 32, immediate = `<n>`.
If `<n>` == 32, immediate = 0.

`ROR #<n>` type = 0b11, immediate = `<n>`.

`RRX` type = 0b11, immediate = 0.

A6.4.2 Register controlled shifts

These are only available in ARM instructions.

`<type>` is the type of shift to apply to the value read from `<Rm>`. It must be one of:

ASR	Arithmetic shift right, encoded as type = 0b10
LSL	Logical shift left, encoded as type = 0b00
LSR	Logical shift right, encoded as type = 0b01
ROR	Rotate right, encoded as type = 0b11.

The bottom byte of `<Rs>` contains the shift amount.

A6.4.3 Shift operations

```
// DecodeImmShift()
// ================

(SRType, integer) DecodeImmShift(bits(2) type, bits(5) imm5)

    case type of
        when ’00’
            shift_t = SRType_LSL;  shift_n = UInt(imm5);
        when ’01’
            shift_t = SRType_LSR;  shift_n = if imm5 == ’00000’ then 32 else UInt(imm5);
        when ’10’
            shift_t = SRType_ASR;  shift_n = if imm5 == ’00000’ then 32 else UInt(imm5);
        when ’11’
            if imm5 == ’00000’ then
                shift_t = SRType_RRX;  shift_n = 1;
            else
                shift_t = SRType_ROR;  shift_n = UInt(imm5);

    return (shift_t, shift_n);

// DecodeRegShift()
// ================

SRType DecodeRegShift(bits(2) type)
    case type of
        when ’00’  shift_t = SRType_LSL;
        when ’01’  shift_t = SRType_LSR;
        when ’10’  shift_t = SRType_ASR;
        when ’11’  shift_t = SRType_ROR;
    return shift_t;


// Shift()
// =======

bits(N) Shift(bits(N) value, SRType type, integer amount, bit carry_in)
    (result, -) = Shift_C(value, type, amount, carry_in);
    return result;
```

```
// Shift_C()
// =========

(bits(N), bit) Shift_C(bits(N) value, SRType type, integer amount, bit carry_in)
    assert !(type == SRType_RRX && amount != 1);

    if amount == 0 then
        (result, carry_out) = (value, carry_in);
    else
        case type of
            when SRType_LSL
                (result, carry_out) = LSL_C(value, amount);
            when SRType_LSR
                (result, carry_out) = LSR_C(value, amount);
            when SRType_ASR
                (result, carry_out) = ASR_C(value, amount);
            when SRType_ROR
                (result, carry_out) = ROR_C(value, amount);
            when SRType_RRX
                (result, carry_out) = RRX_C(value, carry_in);

    return (result, carry_out);
```

A6.5 Memory accesses

The following addressing modes are commonly permitted for memory access instructions:

Offset addressing

The offset value is added to or subtracted from an address obtained from the base register. The result is used as the address for the memory access. The base register is unaltered.

The assembly language syntax for this mode is:

```
[<Rn>,<offset>]
```

Pre-indexed addressing

The offset value is applied to an address obtained from the base register. The result is used as the address for the memory access, and written back into the base register.

The assembly language syntax for this mode is:

```
[<Rn>,<offset>]!
```

Post-indexed addressing

The address obtained from the base register is used, unaltered, as the address for the memory access. The offset value is applied to the address, and written back into the base register.

The assembly language syntax for this mode is:

```
[<Rn>],<offset>
```

In each case, `<Rn>` is the base register. `<offset>` can be:

- an immediate constant, such as `<imm8>` or `<imm12>`
- an index register, `<Rm>`
- a shifted index register, such as `<Rm>, LSL #<shift>`.

For information about unaligned access, endianness, and exclusive access, see:

- *Alignment support* on page A3-3
- *Endian support* on page A3-5
- *Synchronization and semaphores* on page A3-8

A6.6 Hint Instructions

Two classes of hint instruction exist within the Thumb ISA:

- memory hints
- NOP-compatible hints.

A6.6.1 Memory hints

Some load instructions with Rt == 0b1111 are memory *hints*. Memory hints allow you to provide advance information to memory systems about future memory accesses, without actually loading or storing any data.

`PLD`~~, `PLDW`~~ and `PLI` are the only memory hint instructions currently defined, see *Load byte, memory hints* on page A5-24. For instruction details, see:

- *PLD*~~*, PLDW*~~ *(immediate)* on page A6-176
- *PLD (literal)* on page A6-178
- *PLD (register)* on page A6-180
- *PLI (immediate, literal)* on page A6-182
- *PLI (register)* on page A6-184.

Other memory hints are currently unallocated, see *Load halfword, unallocated memory hints* on page A5-23. The effect of a memory hint insturction is IMPLEMENTATION DEFINED. Unallocated memory hints must be implemented as `NOP`, and software must not use them.

A6.6.2 NOP-compatible hints

Hint instructions which are not associated with memory accesses are part of a separate category of hint instructions known as NOP-compatible hints. NOP-compatible hints provide IMPLEMENTATION DEFINED behavior or act as a NOP. Both 16-bit and 32-bit encodings are reserved:

- For information on the 16-bit encodings see *If-Then, and hints* on page A5-11.
- For information on the 32-bit encodings see *Hint instructions* on page A5-19.

A6.7 Alphabetical list of ARMv7-M Thumb instructions

Every ARMv7-M Thumb instruction is listed in this section. See *Format of instruction descriptions* on page A6-2 for details of the format used.

A6.7.1 ADC (immediate)

Add with Carry (immediate) adds an immediate value and the carry flag value to a register value, and writes the result to the destination register. It can optionally update the condition flags based on the result.

Encoding T1 ARMv7-M

```
ADC{S}<c> <Rd>,<Rn>,#<const>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	i	0	1	0	1	0	S	Rn				0	imm3			Rd				imm8							

```
d = UInt(Rd);  n = UInt(Rn);  setflags = (S == '1');  imm32 = ThumbExpandImm(i:imm3:imm8);
if d IN {13,15} || n IN {13,15} then UNPREDICTABLE;
```

Assembler syntax

```
ADC{S}<c><q>  {<Rd>,} <Rn>, #<const>
```

where:

`S` If present, specifies that the instruction updates the flags. Otherwise, the instruction does not update the flags.

`<c><q>` See *Standard assembler syntax fields* on page A6-7.

`<Rd>` Specifies the destination register. If `<Rd>` is omitted, this register is the same as `<Rn>`.

`<Rn>` Specifies the register that contains the first operand.

`<const>` Specifies the immediate value to be added to the value obtained from `<Rn>`. See *Modified immediate constants in Thumb instructions* on page A5-15 for the range of allowed values.

The pre-UAL syntax `ADC<c>S` is equivalent to `ADCS<c>`.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    (result, carry, overflow) = AddWithCarry(R[n], imm32, APSR.C);
    R[d] = result;
    if setflags then
        APSR.N = result<31>;
        APSR.Z = IsZeroBit(result);
        APSR.C = carry;
        APSR.V = overflow;
```

Exceptions

None.

A6.7.2 ADC (register)

Add with Carry (register) adds a register value, the carry flag value, and an optionally-shifted register value, and writes the result to the destination register. It can optionally update the condition flags based on the result.

Encoding T1

All versions of the Thumb ISA.

```
ADCS <Rdn>,<Rm>                                           Outside IT block.
ADC<c> <Rdn>,<Rm>                                         Inside IT block.
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	0	0	1	0	1	Rm			Rdn		

```
d = UInt(Rdn);  n = UInt(Rdn);  m = UInt(Rm);  setflags = !InITBlock();
(shift_t, shift_n) = (SRType_LSL, 0);
```

Encoding T2

ARMv7-M

```
ADC{S}<c>.W <Rd>,<Rn>,<Rm>{,<shift>}
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	0	1	1	0	1	0	S	Rn				(0)	imm3			Rd				imm2		type		Rm			

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  setflags = (S == '1');
(shift_t, shift_n) = DecodeImmShift(type, imm3:imm2);
if d IN {13,15} || n IN {13,15} || m IN {13,15} then UNPREDICTABLE;
```

Assembler syntax

```
ADC{S}<c><q>  {<Rd>,} <Rn>, <Rm> {,<shift>}
```

where:

`S` If present, specifies that the instruction updates the flags. Otherwise, the instruction does not update the flags.

`<c><q>` See *Standard assembler syntax fields* on page A6-7.

`<Rd>` Specifies the destination register. If `<Rd>` is omitted, this register is the same as `<Rn>`.

`<Rn>` Specifies the register that contains the first operand.

`<Rm>` Specifies the register that is optionally shifted and used as the second operand.

`<shift>` Specifies the shift to apply to the value read from `<Rm>`. If `<shift>` is omitted, no shift is applied and both encodings are permitted. If `<shift>` is specified, only encoding T2 is permitted. The possible shifts and how they are encoded are described in *Shifts applied to a register* on page A6-12.

A special case is that if `ADC<c> <Rd>,<Rn>,<Rd>` is written with `<Rd>` and `<Rn>` both in the range R0-R7, it will be assembled using encoding T2 as though `ADC<c> <Rd>,<Rn>` had been written. To prevent this happening, use the .W qualifier.

The pre-UAL syntax `ADC<c>S` is equivalent to `ADCS<c>`.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    shifted = Shift(R[m], shift_t, shift_n, APSR.C);
    (result, carry, overflow) = AddWithCarry(R[n], shifted, APSR.C);
    R[d] = result;
    if setflags then
        APSR.N = result<31>;
        APSR.Z = IsZeroBit(result);
        APSR.C = carry;
        APSR.V = overflow;
```

Exceptions

None.

A6.7.3 ADD (immediate)

This instruction adds an immediate value to a register value, and writes the result to the destination register. It can optionally update the condition flags based on the result.

Encoding T1 All versions of the Thumb ISA.

`ADDS <Rd>,<Rn>,#<imm3>`	Outside IT block.
`ADD<c> <Rd>,<Rn>,#<imm3>`	Inside IT block.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	1	0	imm3			Rn			Rd		

```
d = UInt(Rd);  n = UInt(Rn);  setflags = !InITBlock();  imm32 = ZeroExtend(imm3, 32);
```

Encoding T2 All versions of the Thumb ISA.

`ADDS <Rdn>,#<imm8>`	Outside IT block.
`ADD<c> <Rdn>,#<imm8>`	Inside IT block.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	0	Rdn			imm8							

```
d = UInt(Rdn);  n = UInt(Rdn);  setflags = !InITBlock();  imm32 = ZeroExtend(imm8, 32);
```

Encoding T3 ARMv7-M

`ADD{S}<c>.W <Rd>,<Rn>,#<const>`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	i	0	1	0	0	0	S	Rn				0	imm3			Rd				imm8							

```
if Rd == '1111' && S == '1' then SEE CMN (immediate);
if Rn == '1101' then SEE ADD (SP plus immediate);
d = UInt(Rd);  n = UInt(Rn);  setflags = (S == '1');  imm32 = ThumbExpandImm(i:imm3:imm8);
if d IN {13,15} || n == 15 then UNPREDICTABLE;
```

Encoding T4 ARMv7-M

`ADDW<c> <Rd>,<Rn>,#<imm12>`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	i	1	0	0	0	0	0	Rn				0	imm3			Rd				imm8							

```
if Rn == '1111' then SEE ADR;
if Rn == '1101' then SEE ADD (SP plus immediate);
d = UInt(Rd);  n = UInt(Rn);  setflags = FALSE;  imm32 = ZeroExtend(i:imm3:imm8, 32);
if d IN {13,15} then UNPREDICTABLE;
```

Assembler syntax

`ADD{S}<c><q>  {<Rd>,} <Rn>, #<const>`	All encodings permitted
`ADDW<c><q>  {<Rd>,} <Rn>, #<const>`	Only encoding T4 permitted

where:

S
If present, specifies that the instruction updates the flags. Otherwise, the instruction does not update the flags.

<c><q>
See *Standard assembler syntax fields* on page A6-7.

<Rd>
Specifies the destination register. If <Rd> is omitted, this register is the same as <Rn>.

<Rn>
Specifies the register that contains the first operand. If the SP is specified for <Rn>, see *ADD (SP plus immediate)* on page A6-26. If the PC is specified for <Rn>, see *ADR* on page A6-30.

<const>
Specifies the immediate value to be added to the value obtained from <Rn>. The range of allowed values is 0-7 for encoding T1, 0-255 for encoding T2 and 0-4095 for encoding T4. See *Modified immediate constants in Thumb instructions* on page A5-15 for the range of allowed values for encoding T3.

When multiple encodings of the same length are available for an instruction, encoding T3 is preferred to encoding T4 (if encoding T4 is required, use the ADDW syntax). Encoding T1 is preferred to encoding T2 if <Rd> is specified and encoding T2 is preferred to encoding T1 if <Rd> is omitted.

The pre-UAL syntax ADD<c>S is equivalent to ADDS<c>.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    (result, carry, overflow) = AddWithCarry(R[n], imm32, '0');
    R[d] = result;
    if setflags then
        APSR.N = result<31>;
        APSR.Z = IsZeroBit(result);
        APSR.C = carry;
        APSR.V = overflow;
```

Exceptions

None.

A6.7.4 ADD (register)

This instruction adds a register value and an optionally-shifted register value, and writes the result to the destination register. It can optionally update the condition flags based on the result.

Encoding T1

All versions of the Thumb ISA.

`ADDS <Rd>,<Rn>,<Rm>`	Outside IT block.
`ADD<c> <Rd>,<Rn>,<Rm>`	Inside IT block.

<table>
<tr><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td>0</td><td>0</td><td>0</td><td>1</td><td>1</td><td>0</td><td>0</td><td colspan="3">Rm</td><td colspan="3">Rn</td><td colspan="3">Rd</td></tr>
</table>

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  setflags = !InITBlock();
(shift_t, shift_n) = (SRType_LSL, 0);
```

Encoding T2

All versions of the Thumb ISA.

`ADD<c> <Rdn>,<Rm>`

<table>
<tr><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td>0</td><td>1</td><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td><td>0</td><td>DN</td><td colspan="4">Rm</td><td colspan="3">Rdn</td></tr>
</table>

```
if (DN:Rdn) == '1101' || Rm == '1101' then SEE ADD (SP plus register);
d = UInt(DN:Rdn);  n = UInt(DN:Rdn);  m = UInt(Rm);  setflags = FALSE;
(shift_t, shift_n) = (SRType_LSL, 0);
if d == 15 && InITBlock() && !LastInITBlock() then UNPREDICTABLE;
if d == 15 && m == 15 then UNPREDICTABLE;
```

Encoding T3

ARMv7-M

`ADD{S}<c>.W <Rd>,<Rn>,<Rm>{,<shift>}`

<table>
<tr><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td>1</td><td>1</td><td>1</td><td>0</td><td>1</td><td>0</td><td>1</td><td>1</td><td>0</td><td>0</td><td>0</td><td>S</td><td colspan="4">Rn</td><td>(0)</td><td colspan="3">imm3</td><td colspan="4">Rd</td><td colspan="2">imm2</td><td colspan="2">type</td><td colspan="4">Rm</td></tr>
</table>

```
if Rd == '1111' && S == '1' then SEE CMN (register);
if Rn == '1101' then SEE ADD (SP plus register);
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  setflags = (S == '1');
(shift_t, shift_n) = DecodeImmShift(type, imm3:imm2);
if d IN {13,15} || n == 15 || m IN {13,15} then UNPREDICTABLE;
```

Assembler syntax

```
ADD{S}<c><q>  {<Rd>,} <Rn>, <Rm> {,<shift>}
```

where:

`S` If present, specifies that the instruction updates the flags. Otherwise, the instruction does not update the flags.

`<c><q>` See *Standard assembler syntax fields* on page A6-7.

`<Rd>` Specifies the destination register. If `<Rd>` is omitted, this register is the same as `<Rn>` and encoding T2 is preferred to encoding T1 if both are available (this can only happen inside an IT block). If `<Rd>` is specified, encoding T1 is preferred to encoding T2.

`<Rn>` Specifies the register that contains the first operand. If the SP is specified for `<Rn>`, see *ADD (SP plus register)* on page A6-28.

`<Rm>` Specifies the register that is optionally shifted and used as the second operand.

`<shift>` Specifies the shift to apply to the value read from `<Rm>`. If `<shift>` is omitted, no shift is applied and all encodings are permitted. If `<shift>` is specified, only encoding T3 is permitted. The possible shifts and how they are encoded are described in *Shifts applied to a register* on page A6-12.

Inside an IT block, if `ADD<c> <Rd>,<Rn>,<Rd>` cannot be assembled using encoding T1, it is assembled using encoding T2 as though `ADD<c> <Rd>,<Rn>` had been written. To prevent this happening, use the `.W` qualifier.

The pre-UAL syntax `ADD<c>S` is equivalent to `ADDS<c>`.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    shifted = Shift(R[m], shift_t, shift_n, APSR.C);
    (result, carry, overflow) = AddWithCarry(R[n], shifted, '0');
    if d == 15 then
        ALUWritePC(result);  // setflags is always FALSE here
    else
        R[d] = result;
        if setflags then
            APSR.N = result<31>;
            APSR.Z = IsZeroBit(result);
            APSR.C = carry;
            APSR.V = overflow;
```

Exceptions

None.

A6.7.5 ADD (SP plus immediate)

This instruction adds an immediate value to the SP value, and writes the result to the destination register.

Encoding T1 All versions of the Thumb ISA.

```
ADD<c> <Rd>,SP,#<imm8>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	0	1	Rd			imm8							

```
d = UInt(Rd);  setflags = FALSE;  imm32 = ZeroExtend(imm8:'00', 32);
```

Encoding T2 All versions of the Thumb ISA.

```
ADD<c> SP,SP,#<imm7>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	0	0	0	0	0	imm7						

```
d = 13;  setflags = FALSE;  imm32 = ZeroExtend(imm7:'00', 32);
```

Encoding T3 ARMv7-M

```
ADD{S}<c>.W <Rd>,SP,#<const>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	i	0	1	0	0	0	S	1	1	0	1	0	imm3			Rd				imm8							

```
if Rd == '1111' && S == '1' then SEE CMN (immediate);
d = UInt(Rd);  setflags = (S == '1');  imm32 = ThumbExpandImm(i:imm3:imm8);
if d == 15 then UNPREDICTABLE;
```

Encoding T4 ARMv7-M

```
ADDW<c> <Rd>,SP,#<imm12>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	i	1	0	0	0	0	0	1	1	0	1	0	imm3			Rd				imm8							

```
d = UInt(Rd);  setflags = FALSE;  imm32 = ZeroExtend(i:imm3:imm8, 32);
if d == 15 then UNPREDICTABLE;
```

Assembler syntax

`ADD{S}<c><q>  {<Rd>,} SP, #<const>`	All encodings permitted
`ADDW<c><q>  {<Rd>,} SP, #<const>`	Only encoding T4 is permitted

where:

`S` If present, specifies that the instruction updates the flags. Otherwise, the instruction does not update the flags.

`<c><q>` See *Standard assembler syntax fields* on page A6-7.

`<Rd>` Specifies the destination register. If `<Rd>` is omitted, this register is SP.

`<const>` Specifies the immediate value to be added to the value obtained from `<Rn>`. Allowed values are multiples of 4 in the range 0-1020 for encoding T1, multiples of 4 in the range 0-508 for encoding T2 and any value in the range 0-4095 for encoding T4. See *Modified immediate constants in Thumb instructions* on page A5-15 for the range of allowed values for encoding T3.

When both 32-bit encodings are available for an instruction, encoding T3 is preferred to encoding T4 (if encoding T4 is required, use the `ADDW` syntax).

The pre-UAL syntax `ADD<c>S` is equivalent to `ADDS<c>`.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    (result, carry, overflow) = AddWithCarry(SP, imm32, ’0’);
    R[d] = result;
        if setflags then
            APSR.N = result<31>;
            APSR.Z = IsZeroBit(result);
            APSR.C = carry;
            APSR.V = overflow;
```

Exceptions

None.

A6.7.6 ADD (SP plus register)

This instruction adds an optionally-shifted register value to the SP value, and writes the result to the destination register.

Encoding T1 All versions of the Thumb ISA.

`ADD<c> <Rdm>, SP, <Rdm>`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	1	0	0	DM	1	1	0	1	Rdm		

```
d = UInt(DM:Rdm);  m = UInt(DM:Rdm);  setflags = FALSE;
(shift_t, shift_n) = (SRType_LSL, 0);
```

Encoding T2 All versions of the Thumb ISA.

`ADD<c> SP,<Rm>`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	1	0	0	1	Rm				1	0	1

```
if Rm == '1101' then SEE encoding T1;
d = 13;  m = UInt(Rm);  setflags = FALSE;
(shift_t, shift_n) = (SRType_LSL, 0);
```

Encoding T3 ARMv7-M

`ADD{S}<c>.W <Rd>,SP,<Rm>{,<shift>}`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	0	1	1	0	0	0	S	1	1	0	1	0	imm3			Rd				imm2		type		Rm			

```
d = UInt(Rd);  m = UInt(Rm);  setflags = (S == '1');
(shift_t, shift_n) = DecodeImmShift(type, imm3:imm2);
if d == 13 && (shift_t != SRType_LSL || shift_n > 3) then UNPREDICTABLE;
if d == 15 || m IN {13,15} then UNPREDICTABLE;
```

Assembler syntax

```
ADD{S}<c><q>  {<Rd>,} SP, <Rm>{, <shift>}
```

where:

S If present, specifies that the instruction updates the flags. Otherwise, the instruction does not update the flags.

<c><q> See *Standard assembler syntax fields* on page A6-7.

<Rd> Specifies the destination register. If <Rd> is omitted, this register is SP.

The use of the PC as <Rd> in encoding T1 is deprecated.

<Rm> Specifies the register that is optionally shifted and used as the second operand.

The use of the SP as <Rm> in encoding T1 is deprecated.

The use of the PC as <Rm> in encoding T1 and encoding T2 is deprecated.

<shift> Specifies the shift to apply to the value read from <Rm>. If <shift> is omitted, no shift is applied and all encodings are permitted. If <shift> is specified, only encoding T3 is permitted. The possible shifts and how they are encoded are described in *Shifts applied to a register* on page A6-12.

If <Rd> is SP or omitted, <shift> is only permitted to be LSL #0, LSL #1, LSL #2 or LSL #3.

The pre-UAL syntax ADD<c>S is equivalent to ADDS<c>.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    shifted = Shift(R[m], shift_t, shift_n, APSR.C);
    (result, carry, overflow) = AddWithCarry(SP, shifted, '0');
    if d == 15 then
        ALUWritePC(result);  // setflags is always FALSE here
    else
        R[d] = result;
        if setflags then
            APSR.N = result<31>;
            APSR.Z = IsZeroBit(result);
            APSR.C = carry;
            APSR.V = overflow;
```

Exceptions

None.

A6.7.7 ADR

Address to Register adds an immediate value to the PC value, and writes the result to the destination register.

Encoding T1 All versions of the Thumb ISA.

`ADR<c> <Rd>,<label>`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	0	0	Rd			imm8							

```
d = UInt(Rd);  imm32 = ZeroExtend(imm8:'00', 32);  add = TRUE;
```

Encoding T2 ARMv7-M.

`ADR<c>.W <Rd>,<label>`	`<label>` before current instruction
`SUB <Rd>,PC,#0`	Special case for zero offset

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	i	1	0	1	0	1	0	1	1	1	1	0	imm3			Rd				imm8							

```
d = UInt(Rd);  imm32 = ZeroExtend(i:imm3:imm8, 32);  add = FALSE;
if d IN {13,15} then UNPREDICTABLE;
```

Encoding T3 ARMv7-M

`ADR<c>.W <Rd>,<label>`	`<label>` after current instruction

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	i	1	0	0	0	0	0	1	1	1	1	0	imm3			Rd				imm8							

```
d = UInt(Rd);  imm32 = ZeroExtend(i:imm3:imm8, 32);  add = TRUE;
if d IN {13,15} then UNPREDICTABLE;
```

Assembler syntax

`ADR<c><q>  <Rd>, <label>`	Normal syntax
`ADD<c><q>  <Rd>, PC, #<const>`	Alternative for encodings T1, T3
`SUB<c><q>  <Rd>, PC, #<const>`	Alternative for encoding T2

where:

`<c><q>` See *Standard assembler syntax fields* on page A6-7.

`<Rd>` Specifies the destination register.

`<label>` Specifies the label of an instruction or literal data item whose address is to be loaded into `<Rd>`. The assembler calculates the required value of the offset from the `Align(PC,4)` value of the `ADR` instruction to this label.

If the offset is positive, encodings T1 and T3 are permitted with `imm32` equal to the offset. Allowed values of the offset are multiples of four in the range 0 to 1020 for encoding T1 and any value in the range 0 to 4095 for encoding T3.

If the offset is negative, encoding T2 is permitted with `imm32` equal to minus the offset. Allowed values of the offset are -4095 to -1.

In the alternative syntax forms:

`<const>` Specifies the offset value for the `ADD` form and minus the offset value for the `SUB` form. Allowed values are multiples of four in the range 0 to 1020 for encoding T1 and any value in the range 0 to 4095 for encodings T2 and T3.

Note

It is recommended that the alternative syntax forms are avoided where possible. However, the only possible syntax for encoding T2 with all immediate bits zero is `SUB<c><q> <Rd>,PC,#0`.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    result = if add then (Align(PC,4) + imm32) else (Align(PC,4) - imm32);
    R[d] = result;
```

Exceptions

None.

A6.7.8 AND (immediate)

This instruction performs a bitwise AND of a register value and an immediate value, and writes the result to the destination register.

Encoding T1 ARMv7-M

```
AND{S}<c> <Rd>,<Rn>,#<const>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	i	0	0	0	0	0	S	Rn				0	imm3			Rd				imm8							

```
if Rd == '1111' && S == '1' then SEE TST (immediate);
d = UInt(Rd);  n = UInt(Rn);  setflags = (S == '1');
(imm32, carry) = ThumbExpandImm_C(i:imm3:imm8, APSR.C);
if d IN {13,15} || n IN {13,15} then UNPREDICTABLE;
```

Assembler syntax

```
AND{S}<c><q>  {<Rd>,} <Rn>, #<const>
```

where:

`S` If present, specifies that the instruction updates the flags. Otherwise, the instruction does not update the flags.

`<c><q>` See *Standard assembler syntax fields* on page A6-7.

`<Rd>` Specifies the destination register. If `<Rd>` is omitted, this register is the same as `<Rn>`.

`<Rn>` Specifies the register that contains the first operand.

`<const>` Specifies the immediate value to be added to the value obtained from `<Rn>`. See *Modified immediate constants in Thumb instructions* on page A5-15 for the range of allowed values.

The pre-UAL syntax `AND<c>S` is equivalent to `ANDS<c>`.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    result = R[n] AND imm32;
    R[d] = result;
        if setflags then
            APSR.N = result<31>;
            APSR.Z = IsZeroBit(result);
            APSR.C = carry;
            // APSR.V unchanged
```

Exceptions

None.

A6.7.9 AND (register)

This instruction performs a bitwise AND of a register value and an optionally-shifted register value, and writes the result to the destination register. It can optionally update the condition flags based on the result.

Encoding T1 All versions of the Thumb ISA.

`ANDS <Rdn>,<Rm>`	Outside IT block.
`AND<c> <Rdn>,<Rm>`	Inside IT block.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	0	0	0	0	0	Rm			Rdn		

```
d = UInt(Rdn);  n = UInt(Rdn);  m = UInt(Rm);  setflags = !InITBlock();
(shift_t, shift_n) = (SRType_LSL, 0);
```

Encoding T2 ARMv7-M

`AND{S}<c>.W <Rd>,<Rn>,<Rm>{,<shift>}`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	0	1	0	0	0	0	S	Rn				(0)	imm3			Rd				imm2		type		Rm			

```
if Rd == '1111' && S == '1' then SEE TST (register);
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  setflags = (S == '1');
(shift_t, shift_n) = DecodeImmShift(type, imm3:imm2);
if d IN {13,15} || n IN {13,15} || m IN {13,15} then UNPREDICTABLE;
```

Assembler syntax

```
AND{S}<c><q>  {<Rd>,} <Rn>, <Rm> {,<shift>}
```

where:

`S` If present, specifies that the instruction updates the flags. Otherwise, the instruction does not update the flags.

`<c><q>` See *Standard assembler syntax fields* on page A6-7.

`<Rd>` Specifies the destination register. If `<Rd>` is omitted, this register is the same as `<Rn>`.

`<Rn>` Specifies the register that contains the first operand.

`<Rm>` Specifies the register that is optionally shifted and used as the second operand.

`<shift>` Specifies the shift to apply to the value read from `<Rm>`. If `<shift>` is omitted, no shift is applied and both encodings are permitted. If `<shift>` is specified, only encoding T2 is permitted. The possible shifts and how they are encoded are described in *Shifts applied to a register* on page A6-12.

A special case is that if `AND<c> <Rd>,<Rn>,<Rd>` is written with `<Rd>` and `<Rn>` both in the range R0-R7, it will be assembled using encoding T2 as though `AND<c> <Rd>,<Rn>` had been written. To prevent this happening, use the .W qualifier.

The pre-UAL syntax `AND<c>S` is equivalent to `ANDS<c>`.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    (shifted, carry) = Shift_C(R[m], shift_t, shift_n, APSR.C);
    result = R[n] AND shifted;
    R[d] = result;
        if setflags then
            APSR.N = result<31>;
            APSR.Z = IsZeroBit(result);
            APSR.C = carry;
            // APSR.V unchanged
```

Exceptions

None.

A6.7.10 ASR (immediate)

Arithmetic Shift Right (immediate) shifts a register value right by an immediate number of bits, shifting in copies of its sign bit, and writes the result to the destination register. It can optionally update the condition flags based on the result.

Encoding T1 All versions of the Thumb ISA.

`ASRS <Rd>,<Rm>,#<imm5>`	Outside IT block.
`ASR<c> <Rd>,<Rm>,#<imm5>`	Inside IT block.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	0	imm5					Rm			Rd		

```
d = UInt(Rd);  m = UInt(Rm);  setflags = !InITBlock();
(-, shift_n) = DecodeImmShift('10', imm5);
```

Encoding T2 ARMv7-M

`ASR{S}<c>.W <Rd>,<Rm>,#<imm5>`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	0	1	0	0	1	0	S	1	1	1	1	(0)	imm3			Rd				imm2		1	0	Rm			

```
d = UInt(Rd);  m = UInt(Rm);  setflags = (S == '1');
(-, shift_n) = DecodeImmShift('10', imm3:imm2);
if d IN {13,15} || m IN {13,15} then UNPREDICTABLE;
```

Assembler syntax

```
ASR{S}<c><q>  <Rd>, <Rm>, #<imm5>
```

where:

S If present, specifies that the instruction updates the flags. Otherwise, the instruction does not update the flags.

<c><q> See *Standard assembler syntax fields* on page A6-7.

<Rd> Specifies the destination register.

<Rm> Specifies the register that contains the first operand.

<imm5> Specifies the shift amount, in the range 1 to 32. See *Shifts applied to a register* on page A6-12.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    (result, carry) = Shift_C(R[m], SRType_ASR, shift_n, APSR.C);
    R[d] = result;
        if setflags then
            APSR.N = result<31>;
            APSR.Z = IsZeroBit(result);
            APSR.C = carry;
            // APSR.V unchanged
```

Exceptions

None.

A6.7.11 ASR (register)

Arithmetic Shift Right (register) shifts a register value right by a variable number of bits, shifting in copies of its sign bit, and writes the result to the destination register. The variable number of bits is read from the bottom byte of a register. It can optionally update the condition flags based on the result.

Encoding T1 All versions of the Thumb ISA.

`ASRS <Rdn>,<Rm>`	Outside IT block.
`ASR<c> <Rdn>,<Rm>`	Inside IT block.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	0	0	1	0	0	Rm			Rdn		

```
d = UInt(Rdn);  n = UInt(Rdn);  m = UInt(Rm);  setflags = !InITBlock();
```

Encoding T2 ARMv7-M

`ASR{S}<c>.W <Rd>,<Rn>,<Rm>`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	1	0	0	1	0	S	Rn				1	1	1	1	Rd				0	0	0	0	Rm			

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  setflags = (S == '1');
if d IN {13,15} || n IN {13,15} || m IN {13,15} then UNPREDICTABLE;
```

Assembler syntax

```
ASR{S}<c><q>  <Rd>, <Rn>, <Rm>
```

where:

`S` If present, specifies that the instruction updates the flags. Otherwise, the instruction does not update the flags.

`<c><q>` See *Standard assembler syntax fields* on page A6-7.

`<Rd>` Specifies the destination register.

`<Rn>` Specifies the register that contains the first operand.

`<Rm>` Specifies the register whose bottom byte contains the amount to shift by.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    shift_n = UInt(R[m]<7:0>);
    (result, carry) = Shift_C(R[n], SRType_ASR, shift_n, APSR.C);
    R[d] = result;
    if setflags then
        APSR.N = result<31>;
        APSR.Z = IsZeroBit(result);
        APSR.C = carry;
        // APSR.V unchanged
```

Exceptions

None.

A6.7.12 B

Branch causes a branch to a target address.

Encoding T1 All versions of the Thumb ISA.

`B<c> <label>`

Not allowed in IT block.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	cond				imm8							

```
if cond == '1110' then UNDEFINED;
if cond == '1111' then SEE SVC;
imm32 = SignExtend(imm8:'0', 32);
if InITBlock() then UNPREDICTABLE;
```

Encoding T2 All versions of the Thumb ISA.

`B<c> <label>`

Outside or last in IT block

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	0	imm11										

```
imm32 = SignExtend(imm11:'0', 32);
if InITBlock() && !LastInITBlock() then UNPREDICTABLE;
```

Encoding T3 ARMv7-M

`B<c>.W <label>`

Not allowed in IT block.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	S	cond				imm6						1	0	J1	0	J2	imm11										

Encoding T4 ARMv7-M

`B<c>.W <label>`

Outside or last in IT block

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	S	imm10										1	0	J1	1	J2	imm11										

```
I1 = NOT(J1 EOR S);  I2 = NOT(J2 EOR S);  imm32 = SignExtend(S:I1:I2:imm10:imm11:'0', 32);
if InITBlock() && !LastInITBlock() then UNPREDICTABLE;
```

Assembler syntax

```
B<c><q>  <label>
```

where:

`<c><q>` See *Standard assembler syntax fields* on page A6-7.

Note

Encodings T1 and T3 are conditional in their own right, and do not require an `IT` instruction to make them conditional.

For encodings T1 and T3, `<c>` is not allowed to be `AL` or omitted. The 4-bit encoding of the condition is placed in the instruction and not in a preceding `IT` instruction, and the instruction is not allowed to be in an IT block. As a result, encodings T1 and T2 are never both available to the assembler, nor are encodings T3 and T4.

`<label>` Specifies the label of the instruction that is to be branched to. The assembler calculates the required value of the offset from the PC value of the `B` instruction to this label, then selects an encoding that will set `imm32` to that offset.

Allowed offsets are even numbers in the range -256 to 254 for encoding T1, -2048 to 2046 for encoding T2, -1048576 to 1048574 for encoding T3, and -16777216 to 16777214 for encoding T4.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    BranchWritePC(PC + imm32);
```

Exceptions

None.

Related encodings

If the cond field of encoding T3 is '1110' or '1111', a different instruction is encoded. To determine which instruction, see *Branches and miscellaneous control* on page A5-18.

A6.7.13 BFC

Bit Field Clear clears any number of adjacent bits at any position in a register, without affecting the other bits in the register.

Encoding T1 ARMv7-M

```
BFC<c> <Rd>,#<lsb>,#<width>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	(0)	1	1	0	1	1	0	1	1	1	1	0	imm3			Rd				imm2		(0)	msb				

```
d = UInt(Rd);  msbit = UInt(msb);  lsbit = UInt(imm3:imm2);
if d IN {13,15} then UNPREDICTABLE;
```

Assembler syntax

```
BFC<c><q>  <Rd>, #<lsb>, #<width>
```

where:

`<c><q>` See *Standard assembler syntax fields* on page A6-7.

`<Rd>` Specifies the destination register.

`<lsb>` Specifies the least significant bit that is to be cleared, in the range 0 to 31. This determines the required value of `lsbit`.

`<width>` Specifies the number of bits to be cleared, in the range 1 to 32-`<lsb>`. The required value of `msbit` is `<lsb>+<width>-1`.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    if msbit >= lsbit then
        R[d]<msbit:lsbit> = Replicate('0', msbit-lsbit+1);
        // Other bits of R[d] are unchanged
    else
        UNPREDICTABLE;
```

Exceptions

None.

A6.7.14 BFI

Bit Field Insert copies any number of low order bits from a register into the same number of adjacent bits at any position in the destination register.

Encoding T1 ARMv7-M

`BFI<c> <Rd>,<Rn>,#<lsb>,#<width>`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	(0)	1	1	0	1	1	0	Rn				0	imm3			Rd				imm2		(0)	msb				

```
if Rn == '1111' then SEE BFC;
d = UInt(Rd);  n = UInt(Rn);  msbit = UInt(msb);  lsbit = UInt(imm3:imm2);
if d IN {13,15} || n == 13 then UNPREDICTABLE;
```

Assembler syntax

```
BFI<c><q>  <Rd>, <Rn>, #<lsb>, #<width>
```

where:

`<c><q>` See *Standard assembler syntax fields* on page A6-7.

`<Rd>` Specifies the destination register.

`<Rn>` Specifies the source register.

`<lsb>` Specifies the least significant destination bit, in the range 0 to 31. This determines the required value of `lsbit`.

`<width>` Specifies the number of bits to be copied, in the range 1-32-`<lsb>`. The required value of `msbit` is `<lsb>+<width>-1`.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    if msbit >= lsbit then
        R[d]<msbit:lsbit> = R[n]<(msbit-lsbit):0>;
        // Other bits of R[d] are unchanged
    else
        UNPREDICTABLE;
```

Exceptions

None.

A6.7.15 BIC (immediate)

Bit Clear (immediate) performs a bitwise AND of a register value and the complement of an immediate value, and writes the result to the destination register. It can optionally update the condition flags based on the result.

Encoding T1 ARMv7-M

```
BIC{S}<c> <Rd>,<Rn>,#<const>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	i	0	0	0	0	1	S	Rn				0	imm3			Rd				imm8							

```
d = UInt(Rd);  n = UInt(Rn);  setflags = (S == '1');
(imm32, carry) = ThumbExpandImm_C(i:imm3:imm8, APSR.C);
if d IN {13,15} || n IN {13,15} then UNPREDICTABLE;
```

Assembler syntax

```
BIC{S}<c><q>  {<Rd>,} <Rn>, #<const>
```

where:

`S` If present, specifies that the instruction updates the flags. Otherwise, the instruction does not update the flags.

`<c><q>` See *Standard assembler syntax fields* on page A6-7.

`<Rd>` Specifies the destination register. If `<Rd>` is omitted, this register is the same as `<Rn>`.

`<Rn>` Specifies the register that contains the operand.

`<const>` Specifies the immediate value to be added to the value obtained from `<Rn>`. See *Modified immediate constants in Thumb instructions* on page A5-15 for the range of allowed values.

The pre-UAL syntax `BIC<c>S` is equivalent to `BICS<c>`.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    result = R[n] AND NOT(imm32);
    R[d] = result;
        if setflags then
            APSR.N = result<31>;
            APSR.Z = IsZeroBit(result);
            APSR.C = carry;
            // APSR.V unchanged
```

Exceptions

None.

A6.7.16 BIC (register)

Bit Clear (register) performs a bitwise AND of a register value and the complement of an optionally-shifted register value, and writes the result to the destination register. It can optionally update the condition flags based on the result.

Encoding T1

All versions of the Thumb ISA.

`BICS <Rdn>,<Rm>`	Outside IT block.
`BIC<c> <Rdn>,<Rm>`	Inside IT block.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	0	1	1	1	0	Rm			Rdn		

```
d = UInt(Rdn);  n = UInt(Rdn);  m = UInt(Rm);  setflags = !InITBlock();
(shift_t, shift_n) = (SRType_LSL, 0);
```

Encoding T2

ARMv7-M

`BIC{S}<c>.W <Rd>,<Rn>,<Rm>{,<shift>}`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	0	1	0	0	0	1	S	Rn				(0)	imm3			Rd				imm2		type		Rm			

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  setflags = (S == '1');
(shift_t, shift_n) = DecodeImmShift(type, imm3:imm2);
if d IN {13,15} || n IN {13,15} || m IN {13,15} then UNPREDICTABLE;
```

Assembler syntax

```
BIC{S}<c><q>  {<Rd>,} <Rn>, <Rm> {,<shift>}
```

where:

`S` If present, specifies that the instruction updates the flags. Otherwise, the instruction does not update the flags.

`<c><q>` See *Standard assembler syntax fields* on page A6-7.

`<Rd>` Specifies the destination register. If `<Rd>` is omitted, this register is the same as `<Rn>`.

`<Rn>` Specifies the register that contains the first operand.

`<Rm>` Specifies the register that is optionally shifted and used as the second operand.

`<shift>` Specifies the shift to apply to the value read from `<Rm>`. If `<shift>` is omitted, no shift is applied and both encodings are permitted. If `<shift>` is specified, only encoding T2 is permitted. The possible shifts and how they are encoded are described in *Shifts applied to a register* on page A6-12.

The pre-UAL syntax `BIC<c>S` is equivalent to `BICS<c>`.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    (shifted, carry) = Shift_C(R[m], shift_t, shift_n, APSR.C);
    result = R[n] AND NOT(shifted);
    R[d] = result;
        if setflags then
            APSR.N = result<31>;
            APSR.Z = IsZeroBit(result);
            APSR.C = carry;
            // APSR.V unchanged
```

Exceptions

None.

A6.7.17 BKPT

Breakpoint causes a DebugMonitor exception or a debug halt to occur depending on the configuration of the debug support.

Note

BKPT is an unconditional instruction and executes as such both inside and outside an IT instruction block.

Encoding T1 ARMv5T*, ARMv6-M, ARMv7-M M profile specific behavior

```
BKPT #<imm8>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	1	1	0	imm8							

```
imm32 = ZeroExtend(imm8, 32);
// imm32 is for assembly/disassembly only and is ignored by hardware.
```

Assembler syntax

```
BKPT<q>  #<imm8>
```

where:

`<q>` See *Standard assembler syntax fields* on page A6-7.

`<imm8>` Specifies an 8-bit value that is stored in the instruction. This value is ignored by the ARM hardware, but can be used by a debugger to store additional information about the breakpoint.

Operation

```
EncodingSpecificOperations();
BKPTInstrDebugEvent();
```

Exceptions

DebugMonitor.

A6.7.18 BL

Branch with Link (immediate) calls a subroutine at a PC-relative address.

Encoding T1 All versions of the Thumb ISA.

`BL<c> <label>` Outside or last in IT block

<table>
<tr><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td>1</td><td>1</td><td>1</td><td>1</td><td>0</td><td>S</td><td colspan="10">imm10</td><td>1</td><td>1</td><td>J1</td><td>1</td><td>J2</td><td colspan="11">imm11</td></tr>
</table>

```
I1 = NOT(J1 EOR S);  I2 = NOT(J2 EOR S);  imm32 = SignExtend(S:I1:I2:imm10:imm11:'0', 32);
targetInstrSet = CurrentInstrSet();
if InITBlock() && !LastInITBlock() then UNPREDICTABLE;
```

Assembler syntax

`BL<c><q>  <label>`

where:

`<c><q>` See *Standard assembler syntax fields* on page A6-7.

`<label>` Specifies the label of the instruction that is to be branched to.

The assembler calculates the required value of the offset from the PC value of the `BL` instruction to this label, then selects an encoding that will set `imm32` to that offset. Allowed offsets are even numbers in the range -16777216 to 16777214.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    next_instr_addr = PC;
    LR = next_instr_addr<31:1> : ’1’;
    BranchWritePC(PC + imm32);
```

Exceptions

None.

Note

Before the introduction of Thumb-2 technology, J1 and J2 in encodings T1 and T2 were both 1, resulting in a smaller branch range. The instructions could be executed as two separate 16-bit instructions, with the first instruction `instr1` setting LR to `PC + SignExtend(instr1<10:0>:'00000000000', 32)` and the second instruction completing the operation. It is not possible to split the `BL` instruction into two 16-bit instructions in ARMv6-M and ARMv7-M.

A6.7.19 BLX (register)

Branch with Link and Exchange calls a subroutine at an address and instruction set specified by a register. ARMv7-M only supports the Thumb instruction set. An attempt to change the instruction execution state causes an exception.

Encoding T1

ARMv5T*, ARMv6-M, ARMv7-M

`BLX<c> <Rm>` Outside or last in IT block

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	1	1	1	1	Rm				(0)	(0)	(0)

```
m = UInt(Rm);
if m == 15 then UNPREDICTABLE;
if InITBlock() && !LastInITBlock() then UNPREDICTABLE;
```

Assembler syntax

```
BLX<c><q>  <Rm>
```

where:

`<c><q>` See *Standard assembler syntax fields* on page A6-7.

`<Rm>` Specifies the register that contains the branch target address and instruction set selection bit.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    target = R[m];
    next_instr_addr = PC - 2;
    LR = next_instr_addr<31:1> : '1';
    BLXWritePC(target);
```

Exceptions

UsageFault.

A6.7.20 BX

Branch and Exchange causes a branch to an address and instruction set specified by a register. ARMv7-M only supports the Thumb instruction set. An attempt to change the instruction execution state causes an exception.

Encoding T1

All versions of the Thumb ISA.

`BX<c> <Rm>` Outside or last in IT block

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	1	1	1	0	Rm				(0)	(0)	(0)

```
m = UInt(Rm);
if InITBlock() && !LastInITBlock() then UNPREDICTABLE;
```

Assembler syntax

```
BX<c><q>  <Rm>
```

where:

`<c><q>` See *Standard assembler syntax fields* on page A6-7.

`<Rm>` Specifies the register that contains the branch target address and instruction set selection bit.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    BXWritePC(R[m]);
```

Exceptions

UsageFault.

A6.7.21 CBNZ, CBZ

Compare and Branch on Non-Zero and Compare and Branch on Zero compares the value in a register with zero, and conditionally branches forward a constant value. They do not affect the condition flags.

Encoding T1 ARMv7-M

`CB{N}Z <Rn>,<label>` Not allowed in IT block.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	op	0	i	1	imm5					Rn		

```
n = UInt(Rn);  imm32 = ZeroExtend(i:imm5:'0', 32);  nonzero = (op == '1');
if InITBlock() then UNPREDICTABLE;
```

Assembler syntax

```
CB{N}Z<q>  <Rn>, <label>
```

where:

`<q>` See *Standard assembler syntax fields* on page A6-7.

`<Rn>` The first operand register.

`<label>` The label of the instruction that is to be branched to. The assembler calculates the required value of the offset from the PC value of the `CB{N}Z` instruction to this label, then selects an encoding that will set `imm32` to that offset. Allowed offsets are even numbers in the range 0 to 126.

Operation

```
EncodingSpecificOperations();
if nonzero ^ IsZero(R[n]) then
    BranchWritePC(PC + imm32);
```

Exceptions

None.

A6.7.22 CDP, CDP2

Coprocessor Data Processing tells a coprocessor to perform an operation that is independent of ARM registers and memory.

If no coprocessor can execute the instruction, a UsageFault exception is generated.

Encoding T1 ARMv7-M

```
CDP<c> <coproc>,<opc1>,<CRd>,<CRn>,<CRm>,<opc2>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	1	0	opc1				CRn				CRd				coproc				opc2			0	CRm			

```
cp = UInt(coproc);
```

Encoding T2 ARMv7-M

```
CDP2<c> <coproc>,<opc1>,<CRd>,<CRn>,<CRm>,<opc2>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	0	opc1				CRn				CRd				coproc				opc2			0	CRm			

```
cp = UInt(coproc);
```

Assembler syntax

```
CDP{2}<c><q>  <coproc>, #<opc1>, <CRd>, <CRn>, <CRm> {,#<opc2>}
```

where:

`2` If specified, selects the opc0 == 1 form of the encoding. If omitted, selects the opc0 == 0 form.

`<c><q>` See *Standard assembler syntax fields* on page A6-7.

`<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.

`<opc1>` Is a coprocessor-specific opcode, in the range 0 to 15.

`<CRd>` Specifies the destination coprocessor register for the instruction.

`<CRn>` Specifies the coprocessor register that contains the first operand.

`<CRm>` Specifies the coprocessor register that contains the second operand.

`<opc2>` Is a coprocessor-specific opcode in the range 0 to 7. If it is omitted, `<opc2>` is assumed to be 0.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    if !Coproc_Accepted(cp, ThisInstr()) then
        GenerateCoprocessorException();
    else
        Coproc_InternalOperation(cp, ThisInstr());
```

Exceptions

UsageFault.

Notes

Coprocessor fields Only instruction bits<31:24>, bits<11:8>, and bit<4> are architecturally defined. The remaining fields are recommendations.

A6.7.23 CLREX

Clear Exclusive clears the local record of the executing processor that an address has had a request for an exclusive access.

Encoding T1 ARMv7-M

`CLREX<c>`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	0	1	1	(1)	(1)	(1)	(1)	1	0	(0)	0	(1)	(1)	(1)	(1)	0	0	1	0	(1)	(1)	(1)	(1)

```
// No additional decoding required
```

Assembler syntax

`CLREX<c><q>`

where:

`<c><q>` See *Standard assembler syntax fields* on page A6-7.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    ClearExclusiveLocal(ProcessorID());
```

Exceptions

None.

A6.7.24 CLZ

Count Leading Zeros returns the number of binary zero bits before the first binary one bit in a value.

Encoding T1 ARMv7-M

`CLZ<c> <Rd>,<Rm>`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	1	0	1	0	1	1	Rm				1	1	1	1	Rd				1	0	0	0	Rm			

```
if !Consistent(Rm) then UNPREDICTABLE;
d = UInt(Rd);  m = UInt(Rm);
if d IN {13,15} || m IN {13,15} then UNPREDICTABLE;
```

Assembler syntax

```
CLZ<c><q>  <Rd>, <Rm>
```

where:

`<c><q>` See *Standard assembler syntax fields* on page A6-7.

`<Rd>` Specifies the destination register.

`<Rm>` Specifies the register that contains the operand. Its number must be encoded twice in encoding T1, in both the `Rm` and `Rm2` fields.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    result = CountLeadingZeroBits(R[m]);
    R[d] = result<31:0>;
```

Exceptions

None.

A6.7.25 CMN (immediate)

Compare Negative (immediate) adds a register value and an immediate value. It updates the condition flags based on the result, and discards the result.

Encoding T1 ARMv7-M

```
CMN<c> <Rn>,#<const>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	i	0	1	0	0	0	1	Rn				0	imm3			1	1	1	1	imm8							

```
n = UInt(Rn);  imm32 = ThumbExpandImm(i:imm3:imm8);
if n == 15 then UNPREDICTABLE;
```

Assembler syntax

```
CMN<c><q>  <Rn>, #<const>
```

where:

`<c><q>` See *Standard assembler syntax fields* on page A6-7.

`<Rn>` Specifies the register that contains the operand. This register is allowed to be the SP.

`<const>` Specifies the immediate value to be added to the value obtained from `<Rn>`. See *Modified immediate constants in Thumb instructions* on page A5-15 for the range of allowed values.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    (result, carry, overflow) = AddWithCarry(R[n], imm32, '0');
    APSR.N = result<31>;
    APSR.Z = IsZeroBit(result);
    APSR.C = carry;
    APSR.V = overflow;
```

Exceptions

None.

A6.7.26 CMN (register)

Compare Negative (register) adds a register value and an optionally-shifted register value. It updates the condition flags based on the result, and discards the result.

Encoding T1 All versions of the Thumb ISA.

`CMN<c> <Rn>,<Rm>`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	0	1	0	1	1	Rm			Rn		

```
n = UInt(Rn);  m = UInt(Rm);
(shift_t, shift_n) = (SRType_LSL, 0);
```

Encoding T2 ARMv7-M

`CMN<c>.W  <Rn>,<Rm>{,<shift>}`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	0	1	1	0	0	0	1	Rn				(0)	imm3			1	1	1	1	imm2		type		Rm			

```
n = UInt(Rn);  m = UInt(Rm);
(shift_t, shift_n) = DecodeImmShift(type, imm3:imm2);
if n == 15 || m IN {13,15} then UNPREDICTABLE;
```

Assembler syntax

```
CMN<c><q>  <Rn>, <Rm> {,<shift>}
```

where:

`<c><q>` See *Standard assembler syntax fields* on page A6-7.

`<Rn>` Specifies the register that contains the first operand. This register is allowed to be the SP.

`<Rm>` Specifies the register that is optionally shifted and used as the second operand.

`<shift>` Specifies the shift to apply to the value read from `<Rm>`. If `<shift>` is omitted, no shift is applied and both encodings are permitted. If `<shift>` is specified, only encoding T2 is permitted. The possible shifts and how they are encoded are described in *Shifts applied to a register* on page A6-12.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    shifted = Shift(R[m], shift_t, shift_n, APSR.C);
    (result, carry, overflow) = AddWithCarry(R[n], shifted, ’0’);
    APSR.N = result<31>;
    APSR.Z = IsZeroBit(result);
    APSR.C = carry;
    APSR.V = overflow;
```

Exceptions

None.

A6.7.27 CMP (immediate)

Compare (immediate) subtracts an immediate value from a register value. It updates the condition flags based on the result, and discards the result.

Encoding T1 All versions of the Thumb ISA.

```
CMP<c> <Rn>,#<imm8>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	1	Rn			imm8							

```
n = UInt(Rdn);  imm32 = ZeroExtend(imm8, 32);
```

Encoding T2 ARMv7-M

```
CMP<c>.W <Rn>,#<const>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	i	0	1	1	0	1	1	Rn				0	imm3			1	1	1	1	imm8							

```
n = UInt(Rn);  imm32 = ThumbExpandImm(i:imm3:imm8);
if n == 15 then UNPREDICTABLE;
```

Assembler syntax

```
CMP<c><q>  <Rn>, #<const>
```

where:

`<c><q>` See *Standard assembler syntax fields* on page A6-7.

`<Rn>` Specifies the register that contains the operand. This register is allowed to be the SP.

`<const>` Specifies the immediate value to be added to the value obtained from `<Rn>`. The range of allowed values is 0-255 for encoding T1. See *Modified immediate constants in Thumb instructions* on page A5-15 for the range of allowed values for encoding T2.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    (result, carry, overflow) = AddWithCarry(R[n], NOT(imm32), ’1’);
    APSR.N = result<31>;
    APSR.Z = IsZeroBit(result);
    APSR.C = carry;
    APSR.V = overflow;
```

Exceptions

None.

A6.7.28 CMP (register)

Compare (register) subtracts an optionally-shifted register value from a register value. It updates the condition flags based on the result, and discards the result.

Encoding T1

All versions of the Thumb ISA.

`CMP<c> <Rn>,<Rm>`

`<Rn>` and `<Rm>` both from R0-R7

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	0	1	0	1	0	Rm			Rn		

```
n = UInt(Rn);  m = UInt(Rm);
(shift_t, shift_n) = (SRType_LSL, 0);
```

Encoding T2

All versions of the Thumb ISA.

`CMP<c> <Rn>,<Rm>`

`<Rn>` and `<Rm>` not both from R0-R7

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	1	0	1	N	Rm				Rn		

```
n = UInt(N:Rn);  m = UInt(Rm);
(shift_t, shift_n) = (SRType_LSL, 0);
if n < 8 && m < 8 then UNPREDICTABLE;
if n == 15 || m == 15 then UNPREDICTABLE;
```

Encoding T3

ARMv7-M

`CMP<c>.W <Rn>, <Rm> {,<shift>}`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	0	1	1	1	0	1	1	Rn				(0)	imm3			1	1	1	1	imm2		type		Rm			

```
n = UInt(Rn);  m = UInt(Rm);
(shift_t, shift_n) = DecodeImmShift(type, imm3:imm2);
if n == 15 || m IN {13,15} then UNPREDICTABLE;
```

Assembler syntax

```
CMP<c><q>  <Rn>, <Rm> {,<shift>}
```

where:

`<c><q>` See *Standard assembler syntax fields* on page A6-7.

`<Rn>` Specifies the register that contains the first operand. The SP can be used.

`<Rm>` Specifies the register that is optionally shifted and used as the second operand. The SP can be used, but use of the SP is deprecated.

`<shift>` Specifies the shift to apply to the value read from `<Rm>`. If `<shift>` is omitted, no shift is applied and all encodings are permitted. If shift is specified, only encoding T3 is permitted. The possible shifts and how they are encoded are described in *Shifts applied to a register* on page A6-12.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    shifted = Shift(R[m], shift_t, shift_n, APSR.C);
    (result, carry, overflow) = AddWithCarry(R[n], NOT(shifted), ’1’);
    APSR.N = result<31>;
    APSR.Z = IsZeroBit(result);
    APSR.C = carry;
    APSR.V = overflow;
```

Exceptions

None.

A6.7.29 CPS

Change Processor State. The instruction modifies the PRIMASK and FAULTMASK special-purpose register values.

Encoding T1 ARMv6-M, ARMv7-M Enhanced functionality in ARMv7-M.

`CPS<effect> <iflags>` Not allowed in IT block.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	0	1	1	0	0	1	1	im	(0)	(0)	I	F

Note

`CPS` is a system level instruction with ARMv7-M specific behavior. For the complete instruction definition see *CPS* on page B4-2.

A6.7.30 CPY

Copy is a pre-UAL synonym for `MOV` (register).

Assembler syntax

```
CPY  <Rd>, <Rn>
```

This is equivalent to:

```
MOV  <Rd>, <Rn>
```

Exceptions

None.

A6.7.31 DBG

Debug Hint provides a hint to debug trace support and related debug systems. See debug architecture documentation for what use (if any) is made of this instruction.

This is a NOP-compatible hint. See *NOP-compatible hints* on page A6-16 for general hint behavior.

Encoding T1 ARMv7-M

`DBG<c> #<option>`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	0	1	0	(1)	(1)	(1)	(1)	1	0	(0)	0	(0)	0	0	0	1	1	1	1	option			

```
// Any decoding of 'option' is specified by the debug system
```

Assembler syntax

`DBG<c><q>  #<option>`

where:

`<c><q>` See *Standard assembler syntax fields* on page A6-7.

`<option>` Provides extra information about the hint, and is in the range 0 to 15.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    Hint_Debug(option);
```

Exceptions

None.

A6.7.32 DMB

Data Memory Barrier acts as a memory barrier. It ensures that all explicit memory accesses that appear in program order before the `DMB` instruction are observed before any explicit memory accesses that appear in program order after the `DMB` instruction. It does not affect the ordering of any other instructions executing on the processor.

Encoding T1 ARMv6-M, ARMv7-M

`DMB<c> #<option>`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	0	1	1	(1)	(1)	(1)	(1)	1	0	(0)	0	(1)	(1)	(1)	(1)	0	1	0	1	option			

```
// No additional decoding required
```

Assembler syntax

```
DMB<c><q>  {<opt>}
```

where:

`<c><q>` See *Standard assembler syntax fields* on page A6-7.

`<opt>` Specifies an optional limitation on the DMB operation.

`SY` DMB operation ensures ordering of all accesses, encoded as option == '1111'. Can be omitted.

All other encodings of option are reserved. The corresponding instructions execute as system (SY) DMB operations, but software must not rely on this behavior.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    DataMemoryBarrier(option);
```

Exceptions

None.

A6.7.33 DSB

Data Synchronization Barrier acts as a special kind of memory barrier. No instruction in program order after this instruction can execute until this instruction completes. This instruction completes when:

- All explicit memory accesses before this instruction complete.
- All Cache, Branch predictor ~~and TLB~~ maintenance operations before this instruction complete.

Encoding T1 ARMv6-M, ARMv7-M

```
DSB<c> #<option>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	0	1	1	(1)	(1)	(1)	(1)	1	0	(0)	0	(1)	(1)	(1)	(1)	0	1	0	0	option			

```
// No additional decoding required
```

Assembler syntax

```
DSB<c><q>  {<opt>}
```

where:

`<c><q>` See *Standard assembler syntax fields* on page A6-7.

`<opt>` Specifies an optional limitation on the DSB operation. Values are:

`SY` DSB operation ensures completion of all accesses, encoded as option == '1111'. Can be omitted.

All other encodings of option are reserved. The corresponding instructions execute as system (SY) DSB operations, but software must not rely on this behavior.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    DataSynchronizationBarrier(option);
```

Exceptions

None.

A6.7.34 EOR (immediate)

Exclusive OR (immediate) performs a bitwise Exclusive OR of a register value and an immediate value, and writes the result to the destination register. It can optionally update the condition flags based on the result.

Encoding T1 ARMv7-M

```
EOR{S}<c> <Rd>,<Rn>,#<const>
```

<table>
<tr><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td>1</td><td>1</td><td>1</td><td>1</td><td>0</td><td>i</td><td>0</td><td>0</td><td>1</td><td>0</td><td>0</td><td>S</td><td colspan="4">Rn</td><td>0</td><td colspan="3">imm3</td><td colspan="4">Rd</td><td colspan="8">imm8</td></tr>
</table>

```
if Rd == '1111' && S == '1' then SEE TEQ (immediate);
d = UInt(Rd);  n = UInt(Rn);  setflags = (S == '1');
(imm32, carry) = ThumbExpandImm_C(i:imm3:imm8, APSR.C);
if d IN {13,15} || n IN {13,15} then UNPREDICTABLE;
```

Assembler syntax

```
EOR{S}<c><q>  {<Rd>,} <Rn>, #<const>
```

where:

S	If present, specifies that the instruction updates the flags. Otherwise, the instruction does not update the flags.
<c><q>	See *Standard assembler syntax fields* on page A6-7.
<Rd>	Specifies the destination register. If <Rd> is omitted, this register is the same as <Rn>.
<Rn>	Specifies the register that contains the operand.
<const>	Specifies the immediate value to be added to the value obtained from <Rn>. See *Modified immediate constants in Thumb instructions* on page A5-15 for the range of allowed values.

The pre-UAL syntax EOR<c>S is equivalent to EORS<c>.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    result = R[n] EOR imm32;
    R[d] = result;
    if setflags then
        APSR.N = result<31>;
        APSR.Z = IsZeroBit(result);
        APSR.C = carry;
        // APSR.V unchanged
```

Exceptions

None.

A6.7.35 EOR (register)

Exclusive OR (register) performs a bitwise Exclusive OR of a register value and an optionally-shifted register value, and writes the result to the destination register. It can optionally update the condition flags based on the result.

Encoding T1

All versions of the Thumb ISA.

```
EORS <Rdn>,<Rm>                          Outside IT block.
EOR<c> <Rdn>,<Rm>                        Inside IT block.
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	0	0	0	0	1	Rm			Rdn		

```
d = UInt(Rdn);  n = UInt(Rdn);  m = UInt(Rm);  setflags = !InITBlock();
(shift_t, shift_n) = (SRType_LSL, 0);
```

Encoding T2

ARMv7-M

```
EOR{S}<c>.W <Rd>,<Rn>,<Rm>{,<shift>}
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	0	1	0	1	0	0	S	Rn				(0)	imm3			Rd				imm2		type		Rm			

```
if Rd == '1111' && S == '1' then SEE TEQ (register);
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  setflags = (S == '1');
(shift_t, shift_n) = DecodeImmShift(type, imm3:imm2);
if d IN {13,15} || n IN {13,15} || m IN {13,15} then UNPREDICTABLE;
```

Assembler syntax

```
EOR{S}<c><q>  {<Rd>,} <Rn>, <Rm> {,<shift>}
```

where:

`S` If present, specifies that the instruction updates the flags. Otherwise, the instruction does not update the flags.

`<c><q>` See *Standard assembler syntax fields* on page A6-7.

`<Rd>` Specifies the destination register. If `<Rd>` is omitted, this register is the same as `<Rn>`.

`<Rn>` Specifies the register that contains the first operand.

`<Rm>` Specifies the register that is optionally shifted and used as the second operand.

`<shift>` Specifies the shift to apply to the value read from `<Rm>`. If `<shift>` is omitted, no shift is applied and both encodings are permitted. If `<shift>` is specified, only encoding T2 is permitted. The possible shifts and how they are encoded are described in *Shifts applied to a register* on page A6-12.

A special case is that if `EOR<c> <Rd>,<Rn>,<Rd>` is written with `<Rd>` and `<Rn>` both in the range R0-R7, it will be assembled using encoding T2 as though `EOR<c> <Rd>,<Rn>` had been written. To prevent this happening, use the .W qualifier.

The pre-UAL syntax `EOR<c>S` is equivalent to `EORS<c>`.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    (shifted, carry) = Shift_C(R[m], shift_t, shift_n, APSR.C);
    result = R[n] EOR shifted;
    R[d] = result;
    if setflags then
        APSR.N = result<31>;
        APSR.Z = IsZeroBit(result);
        APSR.C = carry;
        // APSR.V unchanged
```

Exceptions

None.

A6.7.36 ISB

Instruction Synchronization Barrier flushes the pipeline in the processor, so that all instructions following the ISB are fetched from cache or memory, after the instruction has been completed. It ensures that the effects of context altering operations, such as changing the ~~ASID, or completed TLB maintenance operations, or branch predictor maintenance operations, as well as all changes to the CP15~~ registers, executed before the ISB instruction are visible to the instructions fetched after the ISB.

In addition, the ISB instruction ensures that any branches that appear in program order after it are always written into the branch prediction logic with the context that is visible after the ISB instruction. This is required to ensure correct execution of the instruction stream.

Encoding T1 ARMv6-M, ARMv7-M

```
ISB<c> #<option>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	0	1	1	(1)	(1)	(1)	(1)	1	0	(0)	0	(1)	(1)	(1)	(1)	0	1	1	0	option			

```
// No additional decoding required
```

Assembler syntax

```
ISB<c><q>  {<opt>}
```

where:

`<c><q>` See *Standard assembler syntax fields* on page A6-7.

`<opt>` Specifies an optional limitation on the ISB operation. Allowed values are:

`SY` Full system ISB operation, encoded as option == '1111'. Can be omitted.

All other encodings of option are RESERVED. The corresponding instructions execute as full system ISB operations, but should not be relied upon by software.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    InstructionSynchronizationBarrier(option);
```

Exceptions

None.

A6.7.37 IT

If Then makes up to four following instructions (the *IT block*) conditional. The conditions for the instructions in the IT block can be the same, or some of them can be the inverse of others.

IT does not affect the condition code flags. Branches to any instruction in the IT block are not permitted, apart from those performed by exception returns.

16-bit instructions in the IT block, other than CMP, CMN and TST, do not set the condition code flags. The AL condition can be specified to get this changed behavior without conditional execution.

Encoding T1 ARMv7-M

IT{x{y{z}}} <firstcond> Not allowed in IT block

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	1	1	1	firstcond				mask			

```
if mask == '0000' then SEE "Related encodings";
if firstcond == '1111' || (firstcond == '1110' && BitCount(mask) != 1) then UNPREDICTABLE;
if InITBlock() then UNPREDICTABLE;
```

Assembler syntax

IT{x{y{z}}}<q> <firstcond>

where:

<x> Specifies the condition for the second instruction in the IT block.

<y> Specifies the condition for the third instruction in the IT block.

<z> Specifies the condition for the fourth instruction in the IT block.

<q> See *Standard assembler syntax fields* on page A6-7.

<firstcond> Specifies the condition for the first instruction in the IT block.

Each of <x>, <y>, and <z> can be either:

T Then. The condition attached to the instruction is <firstcond>.

E Else. The condition attached to the instruction is the inverse of <firstcond>. The condition code is the same as <firstcond>, except that the least significant bit is inverted. E must not be specified if <firstcond> is AL.

The values of `<x>`, `<y>`, and `<z>` determine the value of the `mask` field as shown in Table A6-3.

Table A6-3 Determination of mask [a] field

<x>	<y>	<z>	mask[3]	mask[2]	mask[1]	mask[0]
omitted	omitted	omitted	1	0	0	0
T	omitted	omitted	firstcond[0]	1	0	0
E	omitted	omitted	NOT firstcond[0]	1	0	0
T	T	omitted	firstcond[0]	firstcond[0]	1	0
E	T	omitted	NOT firstcond[0]	firstcond[0]	1	0
T	E	omitted	firstcond[0]	NOT firstcond[0]	1	0
E	E	omitted	NOT firstcond[0]	NOT firstcond[0]	1	0
T	T	T	firstcond[0]	firstcond[0]	firstcond[0]	1
E	T	T	NOT firstcond[0]	firstcond[0]	firstcond[0]	1
T	E	T	firstcond[0]	NOT firstcond[0]	firstcond[0]	1
E	E	T	NOT firstcond[0]	NOT firstcond[0]	firstcond[0]	1
T	T	E	firstcond[0]	firstcond[0]	NOT firstcond[0]	1
E	T	E	NOT firstcond[0]	firstcond[0]	NOT firstcond[0]	1
T	E	E	firstcond[0]	NOT firstcond[0]	NOT firstcond[0]	1
E	E	E	NOT firstcond[0]	NOT firstcond[0]	NOT firstcond[0]	1

[a] Note that at least one bit is always 1 in mask.

See also *ITSTATE* on page A6-10.

Operation

```
EncodingSpecificOperations();
ITSTATE.IT<7:0> = firstcond:mask;
```

Exceptions

None.

Related encodings

If the mask field of encoding T1 is '0000', a different instruction is encoded. To determine which instruction, see *If-Then, and hints* on page A5-11.

A6.7.38 LDC, LDC2 (immediate)

Load Coprocessor loads memory data from a sequence of consecutive memory addresses to a coprocessor. If no coprocessor can execute the instruction, an UsageFault exception is generated.

This is a generic coprocessor instruction. Some of the fields have no functionality defined by the architecture and are free for use by the coprocessor instruction set designer. These fields are the D bit, the CRd field, and in the Unindexed addressing mode only, the imm8 field.

Encoding T1 ARMv7-M

```
LDC{L}<c> <coproc>,<CRd>,[<Rn>{,#+/-<imm>}]
LDC{L}<c> <coproc>,<CRd>,[<Rn>,#+/-<imm>]!
LDC{L}<c> <coproc>,<CRd>,[<Rn>],#+/-<imm>
LDC{L}<c> <coproc>,<CRd>,[<Rn>],<option>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	0	P	U	D	W	1	Rn				CRd				coproc				imm8							

```
if Rn == '1111' then SEE LDC (literal);
if P == '0' && U == '0' && D == '0' && W == '0' then UNDEFINED;
if P == '0' && U == '0' && D == '1' && W == '0' then SEE MRRC, MRRC2;
n = UInt(Rn);  cp = UInt(coproc);  imm32 = ZeroExtend(imm8:'00', 32);
index = (P == '1');  add = (U == '1');  wback = (W == '1');
```

Encoding T2 ARMv7-M

```
LDC2{L}<c> <coproc>,<CRd>,[<Rn>{,#+/-<imm>}]
LDC2{L}<c> <coproc>,<CRd>,[<Rn>,#+/-<imm>]!
LDC2{L}<c> <coproc>,<CRd>,[<Rn>],#+/-<imm>
LDC2{L}<c> <coproc>,<CRd>,[<Rn>],<option>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	0	P	U	D	W	1	Rn				CRd				coproc				imm8							

```
if Rn == '1111' then SEE LDC (literal);
if P == '0' && U == '0' && D == '0' && W == '0' then UNDEFINED;
if P == '0' && U == '0' && D == '1' && W == '0' then SEE MRRC, MRRC2;
n = UInt(Rn);  cp = UInt(coproc);  imm32 = ZeroExtend(imm8:'00', 32);
index = (P == '1');  add = (U == '1');  wback = (W == '1');
```

Assembler syntax

LDC{2}{L}<c><q>	<coproc>,<CRd>,[<Rn>{,#+/-<imm>}]	Offset. P = 1, W = 0.
LDC{2}{L}<c><q>	<coproc>,<CRd>,[<Rn>,#+/-<imm>]!	Pre-indexed. P = 1, W = 1.
LDC{2}{L}<c><q>	<coproc>,<CRd>,[<Rn>],#+/-<imm>	Post-indexed. P = 0, W = 1.
LDC{2}{L}<c><q>	<coproc>,<CRd>,[<Rn>],<option>	Unindexed. P = 0, W = 0, U = 1.

where:

L If specified, selects the D == 1 form of the encoding. If omitted, selects the D == 0 form.

<c><q> See *Standard assembler syntax fields* on page A6-7.

<coproc> The name of the coprocessor. The standard generic coprocessor names are p0, p1, ..., p15.

<CRd> The coprocessor destination register.

<Rn> The base register. This register is allowed to be the SP or PC.

+/- Is + or omitted if the immediate offset is to be added to the base register value (add == TRUE), or – if it is to be subtracted (add == FALSE). #0 and #-0 generate different instructions.

<imm> The immediate offset applied to the value of <Rn> to form the address. Allowed values are multiples of 4 in the range 0-1020. For the offset addressing syntax, <imm> can be omitted, meaning an offset of +0.

<option> An additional instruction option to the coprocessor. An integer in the range 0-255 enclosed in { }. Encoded in imm8.

The pre-UAL syntax LDC<c>L is equivalent to LDCL<c>.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    if !Coproc_Accepted(cp, ThisInstr()) then
        GenerateCoprocessorException();
    else
        offset_addr = if add then (R[n] + imm32) else (R[n] - imm32);
        address = if index then offset_addr else R[n];
        repeat
            Coproc_SendLoadedWord(MemA[address,4], cp, ThisInstr());  address = address + 4;
        until Coproc_DoneLoading(cp, ThisInstr());
        if wback then R[n] = offset_addr;
```

Exceptions

UsageFault, MemManage, BusFault.

A6.7.39 LDC, LDC2 (literal)

Load Coprocessor loads memory data from a sequence of consecutive memory addresses to a coprocessor. If no coprocessor can execute the instruction, a UsageFault exception is generated.

This is a generic coprocessor instruction. The D bit and the CRd field have no functionality defined by the architecture and are free for use by the coprocessor instruction set designer.

Encoding T1 ARMv7-M

`LDC{L}<c> <coproc>,<CRd>,label`

`LDC{L}<c> <coproc>,<CRd>,[PC,#-0]` Special case `LDC{L}<c> <coproc>,<CRd>,[PC],<option>`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	0	P	U	D	W	1	1	1	1	1	CRd				coproc				imm8							

```
if P == '0' && U == '0' && D == '0' && W == '0' then UNDEFINED;
if P == '0' && U == '0' && D == '1' && W == '0' then SEE MRRC, MRRC2;
index = (P == '1');     // Always TRUE in the Thumb instruction set
add = (U == '1');  cp = UInt(coproc);  imm32 = ZeroExtend(imm8:'00', 32);
if W == '1' || P == '0' then UNPREDICTABLE;
```

Encoding T2 ARMv7-M

`LDC2{L}<c> <coproc>,<CRd>,label`

`LDC2{L}<c> <coproc>,<CRd>,[PC,#-0]` Special case `LDC{L}<c> <coproc>,<CRd>,[PC],<option>`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	0	P	U	D	W	1	1	1	1	1	CRd				coproc				imm8							

```
if P == '0' && U == '0' && D == '0' && W == '0' then UNDEFINED;
if P == '0' && U == '0' && D == '1' && W == '0' then SEE MRRC, MRRC2;
index = (P == '1');     // Always TRUE in the Thumb instruction set
add = (U == '1');  cp = UInt(coproc);  imm32 = ZeroExtend(imm8:'00', 32);
if W == '1' || P == '0' then UNPREDICTABLE;
```

Assembler syntax

`LDC{2}{L}<c><q>  <coproc>,<CRd>,label`	Normal form with P = 1, W = 0
`LDC{2}{L}<c><q>  <coproc>,<CRd>,[PC,#-0]`	Alternative form with P = 1, W = 0

where:

`L` If specified, selects the D == 1 form of the encoding. If omitted, selects the D == 0 form.

`<c><q>` See *Standard assembler syntax fields* on page A6-7.

`<coproc>` The name of the coprocessor. The standard generic coprocessor names are p0, p1, ..., p15.

`<CRd>` The coprocessor destination register.

`<label>` The label of the literal data item that is to be loaded into `<Rt>`. The assembler calculates the required value of the offset from the PC value of this instruction to the label. Permitted values of the offset are multiples of 4 in the range -1020 to 1020.

If the offset is zero or positive, `imm32` is equal to the offset and `add == TRUE`.

If the offset is negative, `imm32` is equal to minus the offset and `add == FALSE`.

The alternative syntax permits the addition or subtraction of the offset and the immediate offset to be specified separately, including permitting a subtraction of 0 that cannot be specified using the normal syntax. For more information, see *Use of labels in UAL instruction syntax* on page A4-5.

The pre-UAL syntax `LDC<c>L` is equivalent to `LDCL<c>`.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    if !Coproc_Accepted(cp, ThisInstr()) then
        GenerateCoprocessorException();
    else
        offset_addr = if add then (Align(PC,4) + imm32) else (Align(PC,4) - imm32);
        address = if index then offset_addr else Align(PC,4);
        repeat
            Coproc_SendLoadedWord(MemA[address,4], cp, ThisInstr());  address = address + 4;
        until Coproc_DoneLoading(cp, ThisInstr());
```

Exceptions

UsageFault, MemManage, BusFault.

A6.7.40 LDM / LDMIA / LDMFD

Load Multiple Increment After loads multiple registers from consecutive memory locations using an address from a base register. The sequential memory locations start at this address, and the address just above the last of those locations can optionally be written back to the base register.

The registers loaded can include the PC. If they do, the word loaded for the PC is treated as ~~an address or exception return value and a branch occurs~~. Bit<0> complies with the ARM architecture interworking rules for branches to Thumb state execution and must be 1. If bit<0> is 0, a UsageFault exception occurs.

Encoding T1 All versions of the Thumb ISA.

`LDM<c> <Rn>!,<registers>`	`<Rn>` not included in `<registers>`
`LDM<c> <Rn>,<registers>`	`<Rn>` included in `<registers>`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	1	Rn			register_list							

```
n = UInt(Rn);  registers = '00000000':register_list;  wback = (registers<n> == '0');
if BitCount(registers) < 1 then UNPREDICTABLE;
```

Encoding T2 ARMv7-M

`LDM<c>.W <Rn>{!},<registers>`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	0	0	0	1	0	W	1	Rn				P	M	(0)	register_list												

```
if W == '1' && Rn == '1101' then SEE POP;
n = UInt(Rn);  registers = P:M:'0':register_list;  wback = (W == '1');
if n == 15 || BitCount(registers) < 2 || (P == '1' && M == '1') then UNPREDICTABLE;
if registers<15> == '1' && InITBlock() && !LastInITBlock() then UNPREDICTABLE;
if wback && registers<n> == '1' then UNPREDICTABLE;
```

Assembler syntax

```
LDM<c><q>  <Rn>{!}, <registers>
```

where:

`<c><q>` See *Standard assembler syntax fields* on page A6-7.

`<Rn>` The base register. If it is the SP and ! is specified, the instruction is treated as described in *POP* on page A6-186.

`!` Causes the instruction to write a modified value back to `<Rn>`. If `!` is omitted, the instruction does not change `<Rn>` in this way.

`<registers>` Is a list of one or more registers to be loaded, separated by commas and surrounded by `{` and `}`. The lowest-numbered register is loaded from the lowest memory address, through to the highest-numbered register from the highest memory address. If the PC is specified in the register list, the instruction causes a branch to the address (data) loaded into the PC.

Encoding T2 does not support a list containing only one register. If an `LDMIA` instruction with just one register `<Rt>` in the list is assembled to Thumb and encoding T1 is not available, it is assembled to the equivalent `LDR<c><q> <Rt>,[<Rn>]{,#4}` instruction.

The SP cannot be in the list.

If the PC is in the list, the LR must not be in the list and the instruction must either be outside an IT block or the last instruction in an IT block.

`LDMIA` and `LDMFD` are pseudo-instructions for `LDM`. `LDMFD` refers to its use for popping data from Full Descending stacks.

The pre-UAL syntaxes `LDM<c>IA` and `LDM<c>FD` are equivalent to `LDM<c>`.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    address = R[n];

    for i = 0 to 14
        if registers<i> == '1' then
            R[i] = MemA[address,4];  address = address + 4;
    if registers<15> == '1' then
        LoadWritePC(MemA[address,4]);

    if wback && registers<n> == '0' then R[n] = R[n] + 4*BitCount(registers);
```

Exceptions

UsageFault, MemManage, BusFault.

A6.7.41 LDMDB / LDMEA

Load Multiple Decrement Before (Load Multiple Empty Ascending) loads multiple registers from sequential memory locations using an address from a base register. The sequential memory locations end just below this address, and the address of the first of those locations can optionally be written back to the base register.

The registers loaded can include the PC. If they do, the word loaded for the PC is treated as ~~an address or exception return value and a branch occurs.~~ Bit<0> complies with the ARM architecture interworking rules for branches to Thumb state execution and must be 1. If bit<0> is 0, a UsageFault exception occurs.

Encoding T1 ARMv7-M

`LDMDB<c> <Rn>{!},<registers>`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	0	0	1	0	0	W	1	Rn				P	M	(0)	register_list												

```
n = UInt(Rn);  registers = P:M:'0':register_list;  wback = (W == '1');
if n == 15 || BitCount(registers) < 2 || (P == '1' && M == '1') then UNPREDICTABLE;
if registers<15> == '1' && InITBlock() && !LastInITBlock() then UNPREDICTABLE;
if wback && registers<n> == '1' then UNPREDICTABLE;
```

Assembler syntax

```
LDMDB<c><q>  <Rn>{!}, <registers>
```

where:

`<c><q>` See *Standard assembler syntax fields* on page A6-7.

`<Rn>` The base register. The SP can be used.

`!` Causes the instruction to write a modified value back to `<Rn>`. Encoded as W = 1.

If `!` is omitted, the instruction does not change `<Rn>` in this way. Encoded as W = 0.

`<registers>`

Is a list of one or more registers, separated by commas and surrounded by `{` and `}`. It specifies the set of registers to be loaded. The registers are loaded with the lowest-numbered register from the lowest memory address, through to the highest-numbered register from the highest memory address. If the PC is specified in the register list, the instruction causes a branch to the address (data) loaded into the PC.

Encoding T1 does not support a list containing only one register. If an `LDMDB` instruction with just one register `<Rt>` in the list is assembled to Thumb, it is assembled to the equivalent `LDR<c><q> <Rt>,[<Rn>,#-4]{!}` instruction.

The SP cannot be in the list.

If the PC is in the list, the LR must not be in the list and the instruction must either be outside an IT block or the last instruction in an IT block.

`LDMEA` is a pseudo-instruction for `LDMDB`, referring to its use for popping data from Empty Ascending stacks.

The pre-UAL syntaxes `LDM<c>DB` and `LDM<c>EA` are equivalent to `LDMDB<c>`.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    address = R[n] - 4*BitCount(registers);

    for i = 0 to 14
        if registers<i> == ’1’ then
            R[i] = MemA[address,4];  address = address + 4;
    if registers<15> == ’1’ then
        LoadWritePC(MemA[address,4]);

    if wback && registers<n> == ’0’ then R[n] = R[n] - 4*BitCount(registers);
```

Exceptions

UsageFault, MemManage, BusFault.

A6.7.42 LDR (immediate)

Load Register (immediate) calculates an address from a base register value and an immediate offset, loads a word from memory, and writes it to a register. It can use offset, post-indexed, or pre-indexed addressing. See *Memory accesses* on page A6-15 for information about memory accesses.

The register loaded can be the PC. If it is, the word loaded for the PC is treated as ~~an address or exception return value and a branch occurs.~~ Bit<0> complies with the ARM architecture interworking rules for branches to Thumb state execution and must be 1. If bit<0> is 0, a UsageFault exception occurs.

Encoding T1 All versions of the Thumb ISA.

```
LDR<c> <Rt>, [<Rn>{,#<imm5>}]
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	1	imm5					Rn			Rt		

```
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm5:'00', 32);
index = TRUE;  add = TRUE;  wback = FALSE;
```

Encoding T2 All versions of the Thumb ISA.

```
LDR<c> <Rt>,[SP{,#<imm8>}]
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	1	1	Rt			imm8							

```
t = UInt(Rt);  n = 13;  imm32 = ZeroExtend(imm8:'00', 32);
index = TRUE;  add = TRUE;  wback = FALSE;
```

Encoding T3 ARMv7-M

```
LDR<c>.W <Rt>,[<Rn>{,#<imm12>}]
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	0	1	1	0	1	Rn				Rt				imm12											

```
if Rn == '1111' then SEE LDR (literal);
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm12, 32);
index = TRUE;  add = TRUE;  wback = FALSE;
if t == 15 && InITBlock() && !LastInITBlock() then UNPREDICTABLE;
```

Encoding T4 ARMv7-M

```
LDR<c> <Rt>,[<Rn>,#-<imm8>]

LDR<c> <Rt>,[<Rn>],#+/-<imm8>

LDR<c> <Rt>,[<Rn>,#+/-<imm8>]!
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	0	0	1	0	1	Rn				Rt				1	P	U	W	imm8							

```
if Rn == '1111' then SEE LDR (literal);
if P == '1' && U == '1' && W == '0' then SEE LDRT;
if Rn == '1101' && P == '0' && U == '1' && W == '1' && imm8 == '00000100' then SEE POP;
if P == '0' && W == '0' then UNDEFINED;
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm8, 32);
index = (P == '1');  add = (U == '1');  wback = (W == '1');
if (wback && n == t) || (t == 15 && InITBlock() && !LastInITBlock()) then UNPREDICTABLE;
```

Assembler syntax

`LDR<c><q>  <Rt>, [<Rn> {, #+/-<imm>}]`	Offset: `index==TRUE`, `wback==FALSE`
`LDR<c><q>  <Rt>, [<Rn>, #+/-<imm>]!`	Pre-indexed: `index==TRUE`, `wback==TRUE`
`LDR<c><q>  <Rt>, [<Rn>], #+/-<imm>`	Post-indexed: `index==FALSE`, `wback==TRUE`

where:

`<c><q>` See *Standard assembler syntax fields* on page A6-7.

`<Rt>` Specifies the destination register. This register is allowed to be the SP. It is also allowed to be the PC, provided the instruction is either outside an IT block or the last instruction of an IT block. If it is the PC, it causes a branch to the address (data) loaded into the PC.

`<Rn>` Specifies the base register. This register is allowed to be the SP. If this register is the PC, see *LDR (literal)* on page A6-90.

`+/-` Is + or omitted to indicate that the immediate offset is added to the base register value (`add == TRUE`), or – to indicate that the offset is to be subtracted (`add == FALSE`). Different instructions are generated for `#0` and `#-0`.

`<imm>` Specifies the immediate offset added to or subtracted from the value of `<Rn>` to form the address. Allowed values are multiples of 4 in the range 0-124 for encoding T1, multiples of 4 in the range 0-1020 for encoding T2, any value in the range 0-4095 for encoding T3, and any value in the range 0-255 for encoding T4. For the offset addressing syntax, `<imm>` can be omitted, meaning an offset of 0.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    offset_addr = if add then (R[n] + imm32) else (R[n] - imm32);
    address = if index then offset_addr else R[n];
    data = MemU[address,4];
    if wback then R[n] = offset_addr;
    if t == 15 then
        if address<1:0> == '00' then LoadWritePC(data);  else UNPREDICTABLE;
    else
        R[t] = data;
```

Exceptions

UsageFault, MemManage, BusFault.

A6.7.43 LDR (literal)

Load Register (literal) calculates an address from the PC value and an immediate offset, loads a word from memory, and writes it to a register. See *Memory accesses* on page A6-15 for information about memory accesses.

The register loaded can be the PC. If it is, the word loaded for the PC is treated as ~~an address or exception return value and a branch occurs~~. Bit<0> complies with the ARM architecture interworking rules for branches to Thumb state execution and must be 1. If bit<0> is 0, a UsageFault exception occurs.

Encoding T1 All versions of the Thumb ISA.

```
LDR<c> <Rt>,<label>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	Rt			imm8							

```
t = UInt(Rt);  imm32 = ZeroExtend(imm8:'00', 32);  add = TRUE;
```

Encoding T2 ARMv7-M

```
LDR<c>.W <Rt>,<label>
LDR<c>.W <Rt>,[PC,#-0]
```

Special case

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	0	U	1	0	1	1	1	1	1	Rt				imm12											

```
t = UInt(Rt);  imm32 = ZeroExtend(imm12, 32);  add = (U == '1');
if t == 15 && InITBlock() && !LastInITBlock() then UNPREDICTABLE;
```

Assembler syntax

`LDR<c><q>  <Rt>, <label>`	Normal form
`LDR<c><q>  <Rt>, [PC, #+/-<imm>]`	Alternative form

where:

`<c><q>` See *Standard assembler syntax fields* on page A6-7.

`<Rt>` The destination register. The SP can be used. The PC can be used, provided the instruction is either outside an IT block or the last instruction of an IT block. If the PC is used, the instruction branches to the address (data) loaded into the PC.

`<label>` The label of the literal data item that is to be loaded into `<Rt>`. The assembler calculates the required value of the offset from the PC value of this instruction to the label. Permitted values of the offset are:

Encoding T1	multiples of four in the range 0 to 1020
Encoding T2	any value in the range -4095 to 4095.

If the offset is zero or positive, `imm32` is equal to the offset and `add == TRUE`.

If the offset is negative, imm32 is equal to minus the offset and add == FALSE. Negative offset is not available in encoding T1.

Note

In code examples in this manual, the syntax =<value> is used for the label of a memory word whose contents are constant and equal to <value>. The actual syntax for such a label is assembler-dependent.

The alternative syntax permits the addition or subtraction of the offset and the immediate offset to be specified separately, including permitting a subtraction of 0 that cannot be specified using the normal syntax. For more information, see *Use of labels in UAL instruction syntax* on page A4-5.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    base = Align(PC,4);
    address = if add then (base + imm32) else (base - imm32);
    data = MemU[address,4];
    if t == 15 then
        if address<1:0> == '00' then LoadWritePC(data);  else UNPREDICTABLE;
    else
        R[t] = data;
```

Exceptions

UsageFault, MemManage, BusFault.

A6.7.44 LDR (register)

Load Register (register) calculates an address from a base register value and an offset register value, loads a word from memory, and writes it to a register. The offset register value can be shifted left by 0, 1, 2, or 3 bits. See *Memory accesses* on page A6-15 for information about memory accesses.

The register loaded can be the PC. If it is, the word loaded for the PC is treated as ~~an address or exception return value and a branch occurs.~~ Bit<0> complies with the ARM architecture interworking rules for branches to Thumb state execution and must be 1. If bit<0> is 0, a UsageFault exception occurs.

Encoding T1 All versions of the Thumb ISA.

```
LDR<c> <Rt>,[<Rn>,<Rm>]
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	0	0	Rm			Rn			Rt		

```
t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);
index = TRUE;  add = TRUE;  wback = FALSE;
(shift_t, shift_n) = (SRType_LSL, 0);
```

Encoding T2 ARMv7-M

```
LDR<c>.W <Rt>,[<Rn>,<Rm>{,LSL #<imm2>}]
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	0	0	1	0	1	Rn				Rt				0	0	0	0	0	0	imm2		Rm			

```
if Rn == '1111' then SEE LDR (literal);
t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);
index = TRUE;  add = TRUE;  wback = FALSE;
(shift_t, shift_n) = (SRType_LSL, UInt(imm2));
if m IN {13,15} then UNPREDICTABLE;
if t == 15 && InITBlock() && !LastInITBlock() then UNPREDICTABLE;
```

Assembler syntax

```
LDR<c><q>  <Rt>, [<Rn>, <Rm> {, LSL #<shift>}]
```

where:

`<c><q>` See *Standard assembler syntax fields* on page A6-7.

`<Rt>` Specifies the destination register. This register is allowed to be the SP. It is also allowed to be the PC, provided the instruction is either outside an IT block or the last instruction of an IT block. If it is the PC, it causes a branch to the address (data) loaded into the PC.

`<Rn>` Specifies the register that contains the base value. This register is allowed to be the SP.

`<Rm>` Contains the offset that is shifted left and added to the value of `<Rn>` to form the address.

`<shift>` Specifies the number of bits the value from `<Rm>` is shifted left, in the range 0-3. If this option is omitted, a shift by 0 is assumed and both encodings are permitted. If this option is specified, only encoding T2 is permitted.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    offset = Shift(R[m], shift_t, shift_n, APSR.C);
    offset_addr = if add then (R[n] + offset) else (R[n] - offset);
    address = if index then offset_addr else R[n];
    data = MemU[address,4];
    if wback then R[n] = offset_addr;
    if t == 15 then
        if address<1:0> == '00' then LoadWritePC(data);  else UNPREDICTABLE;
    else
        R[t] = data;
```

Exceptions

UsageFault, MemManage, BusFault.

A6.7.45 LDRB (immediate)

Load Register Byte (immediate) calculates an address from a base register value and an immediate offset, loads a byte from memory, zero-extends it to form a 32-bit word, and writes it to a register. It can use offset, post-indexed, or pre-indexed addressing. See *Memory accesses* on page A6-15 for information about memory accesses.

Encoding T1 All versions of the Thumb ISA.

```
LDRB<c> <Rt>,[<Rn>{,#<imm5>}]
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	imm5					Rn			Rt		

```
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm5, 32);
index = TRUE;  add = TRUE;  wback = FALSE;
```

Encoding T2 ARMv7-M

```
LDRB<c>.W <Rt>,[<Rn>{,#<imm12>}]
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	0	1	0	0	1	Rn				Rt				imm12											

```
if Rt == '1111' then SEE PLD;
if Rn == '1111' then SEE LDRB (literal);
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm12, 32);
index = TRUE;  add = TRUE;  wback = FALSE;
if t == 13 then UNPREDICTABLE;
```

Encoding T3 ARMv7-M

```
LDRB<c> <Rt>,[<Rn>,#-<imm8>]

LDRB<c> <Rt>,[<Rn>],#+/-<imm8>

LDRB<c> <Rt>,[<Rn>,#+/-<imm8>]!
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	0	0	0	0	1	Rn				Rt				1	P	U	W	imm8							

```
if Rt == '1111' && P == '1' && U == '0' && W == '0' then SEE PLD;
if Rn == '1111' then SEE LDRB (literal);
if P == '1' && U == '1' && W == '0' then SEE LDRBT;
if P == '0' && W == '0' then UNDEFINED;
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm8, 32);
index = (P == '1');  add = (U == '1');  wback = (W == '1');
if t IN {13,15} || (wback && n == t) then UNPREDICTABLE;
```

Assembler syntax

`LDRB<c><q>  <Rt>, [<Rn> {, #+/-<imm>}]`	Offset: `index==TRUE`, `wback==FALSE`
`LDRB<c><q>  <Rt>, [<Rn>, #+/-<imm>]!`	Pre-indexed: `index==TRUE`, `wback==TRUE`
`LDRB<c><q>  <Rt>, [<Rn>], #+/-<imm>`	Post-indexed: `index==FALSE`, `wback==TRUE`

where:

`<c><q>` See *Standard assembler syntax fields* on page A6-7.

`<Rt>` Specifies the destination register.

`<Rn>` Specifies the base register. This register is allowed to be the SP. If this register is the PC, see *LDRB (literal)* on page A6-96.

`+/-` Is + or omitted to indicate that the immediate offset is added to the base register value (`add == TRUE`), or – to indicate that the offset is to be subtracted (`add == FALSE`). Different instructions are generated for `#0` and `#-0`.

`<imm>` Specifies the immediate offset added to or subtracted from the value of `<Rn>` to form the address. The range of allowed values is 0-31 for encoding T1, 0-4095 for encoding T2, and 0-255 for encoding T3. For the offset addressing syntax, `<imm>` can be omitted, meaning an offset of 0.

The pre-UAL syntax `LDR<c>B` is equivalent to `LDRB<c>`.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    offset_addr = if add then (R[n] + imm32) else (R[n] - imm32);
    address = if index then offset_addr else R[n];
    R[t] = ZeroExtend(MemU[address,1], 32);
    if wback then R[n] = offset_addr;
```

Exceptions

MemManage, BusFault.

A6.7.46 LDRB (literal)

Load Register Byte (literal) calculates an address from the PC value and an immediate offset, loads a byte from memory, zero-extends it to form a 32-bit word, and writes it to a register. See *Memory accesses* on page A6-15 for information about memory accesses.

Encoding T1 ARMv7-M

```
LDRB<c> <Rt>,<label>
LDRB<c> <Rt>,[PC,#-0]
```

Special case

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	0	U	0	0	1	1	1	1	1	Rt				imm12											

```
if Rt == '1111' then SEE PLD;
t = UInt(Rt);  imm32 = ZeroExtend(imm12, 32);  add = (U == '1');
if t == 13 then UNPREDICTABLE;
```

Assembler syntax

`LDRB<c><q>  <Rt>, <label>`	Normal form
`LDRB<c><q>  <Rt>, [PC, #+/-<imm>]`	Alternative form

where:

`<c><q>` See *Standard assembler syntax fields* on page A6-7.

`<Rt>` The destination register.

`<label>` The label of the literal data item that is to be loaded into `<Rt>`. The assembler calculates the required value of the offset from the PC value of this instruction to the label. Permitted values of the offset are -4095 to 4095.

If the offset is zero or positive, `imm32` is equal to the offset and `add == TRUE`.

If the offset is negative, `imm32` is equal to minus the offset and `add == FALSE`.

The alternative syntax permits the addition or subtraction of the offset and the immediate offset to be specified separately, including permitting a subtraction of 0 that cannot be specified using the normal syntax. For more information, see *Use of labels in UAL instruction syntax* on page A4-5.

The pre-UAL syntax `LDR<c>B` is equivalent to `LDRB<c>`.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    base = Align(PC,4);
    address = if add then (base + imm32) else (base - imm32);
    R[t] = ZeroExtend(MemU[address,1], 32);
```

Exceptions

MemManage, BusFault.

A6.7.47 LDRB (register)

Load Register Byte (register) calculates an address from a base register value and an offset register value, loads a byte from memory, zero-extends it to form a 32-bit word, and writes it to a register. The offset register value can be shifted left by 0, 1, 2, or 3 bits. See *Memory accesses* on page A6-15 for information about memory accesses.

Encoding T1 All versions of the Thumb ISA.

```
LDRB<c> <Rt>,[<Rn>,<Rm>]
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	0	Rm			Rn			Rt		

```
t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);
index = TRUE;  add = TRUE;  wback = FALSE;
(shift_t, shift_n) = (SRType_LSL, 0);
```

Encoding T2 ARMv7-M

```
LDRB<c>.W <Rt>,[<Rn>,<Rm>{,LSL #<imm2>}]
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	0	0	0	0	1	Rn				Rt				0	0	0	0	0	0	imm2		Rm			

```
if Rt == '1111' then SEE PLD;
if Rn == '1111' then SEE LDRB (literal);
t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);
index = TRUE;  add = TRUE;  wback = FALSE;
(shift_t, shift_n) = (SRType_LSL, UInt(imm2));
if t == 13 || m IN {13,15} then UNPREDICTABLE;
```

Assembler syntax

```
LDRB<c><q>  <Rt>, [<Rn>, <Rm> {, LSL #<shift>}]
```

where:

`<c><q>` See *Standard assembler syntax fields* on page A6-7.

`<Rt>` The destination register.

`<Rn>` Specifies the register that contains the base value. This register is allowed to be the SP.

`<Rm>` Contains the offset that is shifted left and added to the value of `<Rn>` to form the address.

`<shift>` Specifies the number of bits the value from `<Rm>` is shifted left, in the range 0-3. If this option is omitted, a shift by 0 is assumed and both encodings are permitted. If this option is specified, only encoding T2 is permitted.

The pre-UAL syntax `LDR<c>B` is equivalent to `LDRB<c>`.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    offset = Shift(R[m], shift_t, shift_n, APSR.C);
    offset_addr = if add then (R[n] + offset) else (R[n] - offset);
    address = if index then offset_addr else R[n];
    R[t] = ZeroExtend(MemU[address,1],32);
```

Exceptions

MemManage, BusFault.

A6.7.48 LDRBT

Load Register Byte Unprivileged calculates an address from a base register value and an immediate offset, loads a byte from memory, zero-extends it to form a 32-bit word, and writes it to a register. See *Memory accesses* on page A6-15 for information about memory accesses.

The memory access is restricted as if the processor were running unprivileged. (This makes no difference if the processor is actually running unprivileged.)

Encoding T1 ARMv7-M

```
LDRBT<c> <Rt>,[<Rn>,#<imm8>]
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	0	0	0	0	1	Rn				Rt				1	1	1	0	imm8							

```
if Rn == '1111' then SEE LDRB (literal);
t = UInt(Rt);  n = UInt(Rn);  postindex = FALSE;  add = TRUE;
register_form = FALSE;  imm32 = ZeroExtend(imm8, 32);
if t IN {13,15} then UNPREDICTABLE;
```

Assembler syntax

```
LDRBT<c><q>  <Rt>, [<Rn> {, #<imm>}]
```

where:

`<c><q>` See *Standard assembler syntax fields* on page A6-7.

`<Rt>` Specifies the destination register.

`<Rn>` Specifies the base register. This register is allowed to be the SP.

`<imm>` Specifies the immediate offset added to the value of `<Rn>` to form the address. The range of allowed values is 0-255. `<imm>` can be omitted, meaning an offset of 0.

The pre-UAL syntax `LDR<c>BT` is equivalent to `LDRBT<c>`.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    address = R[n] + imm32;
    R[t] = ZeroExtend(MemU_unpriv[address,1],32);
```

Exceptions

MemManage, BusFault.

A6.7.49 LDRD (immediate)

Load Register Dual (immediate) calculates an address from a base register value and an immediate offset, loads two words from memory, and writes them to two registers. It can use offset, post-indexed, or pre-indexed addressing. See *Memory accesses* on page A6-15 for information about memory accesses.

Encoding T1 ARMv7-M

```
LDRD<c> <Rt>,<Rt2>,[<Rn>{,#+/-<imm8>}]
LDRD<c> <Rt>,<Rt2>,[<Rn>],#+/-<imm8>
LDRD<c> <Rt>,<Rt2>,[<Rn>,#+/-<imm8>]!
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	0	0	P	U	1	W	1	Rn				Rt				Rt2				imm8							

```
if P == '0' && W == '0' then SEE "Related encodings";
if Rn == '1111' then SEE LDRD (literal);
t = UInt(Rt);  t2 = UInt(Rt2);  n = UInt(Rn);  imm32 = ZeroExtend(imm8:'00', 32);
index = (P == '1');  add = (U == '1');  wback = (W == '1');
if wback && (n == t || n == t2) then UNPREDICTABLE;
if t IN {13,15} || t2 IN {13,15} || t == t2 then UNPREDICTABLE;
```

Related encodings See *Load/store dual or exclusive, table branch* on page A5-21

Assembler syntax

`LDRD<c><q> <Rt>,<Rt2>,[<Rn>{,#+/-<imm>}]`	Offset: `index==TRUE`, `wback==FALSE`
`LDRD<c><q> <Rt>,<Rt2>,[<Rn>,#+/-<imm>]!`	Pre-indexed: `index==TRUE`, `wback==TRUE`
`LDRD<c><q> <Rt>,<Rt2>,[<Rn>],#+/-<imm>`	Post-indexed: `index==FALSE`, `wback==TRUE`

where:

`<c><q>` See *Standard assembler syntax fields* on page A6-7.

`<Rt>` Specifies the first destination register.

`<Rt2>` Specifies the second destination register.

`<Rn>` Specifies the base register. This register is allowed to be the SP. In the offset addressing form of the syntax, it is also allowed to be the PC.

`+/-` Is + or omitted to indicate that the immediate offset is added to the base register value (`add == TRUE`), or – to indicate that the offset is to be subtracted (`add == FALSE`). Different instructions are generated for `#0` and `#-0`.

`<imm>` Specifies the immediate offset added to or subtracted from the value of `<Rn>` to form the address. Allowed values are multiples of 4 in the range 0-1020. For the offset addressing syntax, `<imm>` can be omitted, meaning an offset of 0.

The pre-UAL syntax `LDR<c>D` is equivalent to `LDRD<c>`.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    offset_addr = if add then (R[n] + imm32) else (R[n] - imm32);
    address = if index then offset_addr else R[n];
    R[t] = MemA[address,4];
    R[t2] = MemA[address+4,4];
    if wback then R[n] = offset_addr;
```

Exceptions

UsageFault, MemManage, BusFault.

A6.7.50 LDRD (literal)

Load Register Dual (literal) calculates an address from the PC value and an immediate offset, loads two words from memory, and writes them to two registers. See *Memory accesses* on page A6-15 for information about memory accesses.

Note

For the M profile, the PC value must be word-aligned, otherwise the behavior of the instruction is UNPREDICTABLE.

Encoding T1 ARMv7-M

```
LDRD<c> <Rt>,<Rt2>,<label>
LDRD<c> <Rt>,<Rt2>,[PC,#-0]
```

Special case

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	0	0	P	U	1	~~(0)~~	1	1	1	1	1	Rt				Rt2				imm8							

```
if P == '0' && W == '0' then SEE "Related encodings";
t = UInt(Rt);  t2 = UInt(Rt2);
imm32 = ZeroExtend(imm8:'00', 32);  add = (U == '1');
if t IN {13,15} || t2 IN {13,15} || t == t2 then UNPREDICTABLE;
if W == '1' then UNPREDICTABLE;
```

Related encodings See *Load/store dual or exclusive, table branch* on page A5-21

Assembler syntax

`LDRD<c><q>  <Rt>, <Rt2>, <label>`	Normal form
`LDRD<c><q>  <Rt>, <Rt2>, [PC, #+/-<imm>]`	Alternative form

where:

`<c><q>` See *Standard assembler syntax fields* on page A6-7.

`<Rt>` The first destination register.

`<Rt2>` The second destination register.

`<label>` The label of the literal data item that is to be loaded into `<Rt>`. The assembler calculates the required value of the offset from the PC value of this instruction to the label. Permitted values of the offset are multiples of 4 in the range -1020 to 1020.

If the offset is zero or positive, `imm32` is equal to the offset and `add == TRUE`.

If the offset is negative, `imm32` is equal to minus the offset and `add == FALSE`.

The alternative syntax permits the addition or subtraction of the offset and the immediate offset to be specified separately, including permitting a subtraction of 0 that cannot be specified using the normal syntax. For more information, see *Use of labels in UAL instruction syntax* on page A4-5.

The pre-UAL syntax `LDR<c>D` is equivalent to `LDRD<c>`.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    if PC<1:0> != '00' then UNPREDICTABLE;
    address = if add then (PC + imm32) else (PC - imm32);
    R[t] = MemA[address,4];
    R[t2] = MemA[address+4,4];
```

Exceptions

MemManage, BusFault.

A6.7.51 LDREX

Load Register Exclusive calculates an address from a base register value and an immediate offset, loads a word from memory, writes it to a register and:

- if the address has the Shareable Memory attribute, marks the physical address as exclusive access for the executing processor in a global monitor
- causes the executing processor to indicate an active exclusive access in the local monitor.

See *Memory accesses* on page A6-15 for information about memory accesses.

Encoding T1 ARMv7-M

`LDREX<c> <Rt>,[<Rn>{,#<imm8>}]`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	0	0	0	0	1	0	1	Rn				Rt				(1)	(1)	(1)	(1)	imm8							

```
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm8:'00', 32);
if t IN {13,15} || n == 15 then UNPREDICTABLE;
```

Assembler syntax

```
LDREX<c><q>  <Rt>, [<Rn> {,#<imm>}]
```

where:

`<c><q>` See *Standard assembler syntax fields* on page A6-7.

`<Rt>` Specifies the destination register.

`<Rn>` Specifies the base register. This register is allowed to be the SP.

`<imm>` Specifies the immediate offset added to the value of `<Rn>` to form the address. Allowed values are multiples of 4 in the range 0-1020. `<imm>` can be omitted, meaning an offset of 0.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    address = R[n] + imm32;
    SetExclusiveMonitors(address,4);
    R[t] = MemA[address,4];
```

Exceptions

UsageFault, MemManage, BusFault.

A6.7.52 LDREXB

Load Register Exclusive Byte derives an address from a base register value, loads a byte from memory, zero-extends it to form a 32-bit word, writes it to a register and:

- if the address has the Shareable Memory attribute, marks the physical address as exclusive access for the executing processor in a global monitor
- causes the executing processor to indicate an active exclusive access in the local monitor.

See *Memory accesses* on page A6-15 for information about memory accesses.

Encoding T1 ARMv7

```
LDREXB<c> <Rt>, [<Rn>]
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	0	0	0	1	1	0	1	Rn				Rt				(1)	(1)	(1)	(1)	0	1	0	0	(1)	(1)	(1)	(1)

```
t = UInt(Rt);  n = UInt(Rn);
if t IN {13,15} || n == 15 then UNPREDICTABLE;
```

Assembler syntax

```
LDREXB<c><q>  <Rt>, [<Rn>]
```

where:

`<c><q>` See *Standard assembler syntax fields* on page A6-7.

`<Rt>` Specifies the destination register.

`<Rn>` Specifies the base register. This register is allowed to be the SP.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    address = R[n];
    SetExclusiveMonitors(address,1);
    R[t] = ZeroExtend(MemA[address,1], 32);
```

Exceptions

MemManage, BusFault.

A6.7.53 LDREXH

Load Register Exclusive Halfword derives an address from a base register value, loads a halfword from memory, zero-extends it to form a 32-bit word, writes it to a register and:

- if the address has the Shareable Memory attribute, marks the physical address as exclusive access for the executing processor in a global monitor
- causes the executing processor to indicate an active exclusive access in the local monitor.

See *Memory accesses* on page A6-15 for information about memory accesses.

Encoding T1 ARMv7

`LDREXH<c> <Rt>, [<Rn>]`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	0	0	0	1	1	0	1	Rn				Rt				(1)	(1)	(1)	(1)	0	1	0	1	(1)	(1)	(1)	(1)

```
t = UInt(Rt);  n = UInt(Rn);
if t IN {13,15} || n == 15 then UNPREDICTABLE;
```

Assembler syntax

```
LDREXH<c><q>  <Rt>, [<Rn>]
```

where:

`<c><q>` See *Standard assembler syntax fields* on page A6-7.

`<Rt>` Specifies the destination register.

`<Rn>` Specifies the base register. This register is allowed to be the SP.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    address = R[n];
    SetExclusiveMonitors(address,2);
    R[t] = ZeroExtend(MemA[address,2], 32);
```

Exceptions

UsageFault, MemManage, BusFault.

A6.7.54 LDRH (immediate)

Load Register Halfword (immediate) calculates an address from a base register value and an immediate offset, loads a halfword from memory, zero-extends it to form a 32-bit word, and writes it to a register. It can use offset, post-indexed, or pre-indexed addressing. See *Memory accesses* on page A6-15 for information about memory accesses.

Encoding T1 All versions of the Thumb ISA.

```
LDRH<c> <Rt>,[<Rn>{,#<imm5>}]
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	0	1	imm5					Rn			Rt		

```
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm5:’0’, 32);
index = TRUE;  add = TRUE;  wback = FALSE;
```

Encoding T2 ARMv7-M

```
LDRH<c>.W <Rt>,[<Rn>{,#<imm12>}]
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	0	1	0	1	1	Rn				Rt				imm12											

```
if Rt == ’1111’ then SEE "Unallocated memory hints";
if Rn == ’1111’ then SEE LDRH (literal);
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm12, 32);
index = TRUE;  add = TRUE;  wback = FALSE;
if t == 13 then UNPREDICTABLE;
```

Encoding T3 ARMv7-M

```
LDRH<c> <Rt>,[<Rn>,#-<imm8>]
LDRH<c> <Rt>,[<Rn>],#+/-<imm8>
LDRH<c> <Rt>,[<Rn>,#+/-<imm8>]!
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	0	0	0	1	1	Rn				Rt				1	P	U	W	imm8							

```
if Rn == '1111' then SEE LDRH (literal);
if Rt == '1111' && P == '1' && U == '0' && W == '0' then SEE "Unallocated memory hints";
if P == '1' && U == '1' && W == '0' then SEE LDRHT;
if P == '0' && W == '0' then UNDEFINED;
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm8, 32);
index = (P == '1');  add = (U == '1');  wback = (W == '1');
if t IN {13,15} || (wback && n == t) then UNPREDICTABLE;
```

Assembler syntax

`LDRH<c><q>  <Rt>, [<Rn> {, #+/-<imm>}]`	Offset: `index==TRUE`, `wback==FALSE`
`LDRH<c><q>  <Rt>, [<Rn>, #+/-<imm>]!`	Pre-indexed: `index==TRUE`, `wback==TRUE`
`LDRH<c><q>  <Rt>, [<Rn>], #+/-<imm>`	Post-indexed: `index==FALSE`, `wback==TRUE`

where:

`<c><q>` See *Standard assembler syntax fields* on page A6-7.

`<Rt>` Specifies the destination register.

`<Rn>` Specifies the base register. This register is allowed to be the SP. If this register is the PC, see *LDRH (literal)* on page A6-112.

`+/-` Is + or omitted to indicate that the immediate offset is added to the base register value (`add == TRUE`), or – to indicate that the offset is to be subtracted (`add == FALSE`). Different instructions are generated for `#0` and `#-0`.

`<imm>` Specifies the immediate offset added to or subtracted from the value of `<Rn>` to form the address. Allowed values are multiples of 2 in the range 0-62 for encoding T1, any value in the range 0-4095 for encoding T2, and any value in the range 0-255 for encoding T3. For the offset addressing syntax, `<imm>` can be omitted, meaning an offset of 0.

The pre-UAL syntax `LDR<c>H` is equivalent to `LDRH<c>`.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    offset_addr = if add then (R[n] + imm32) else (R[n] - imm32);
    address = if index then offset_addr else R[n];
    data = MemU[address,2];
    if wback then R[n] = offset_addr;
    R[t] = ZeroExtend(data, 32);
```

Exceptions

UsageFault, MemManage, BusFault.

Unallocated memory hints

If the Rt field is '1111' in encoding T2, or if the Rt field and P, U, and W bits in encoding T3 are '1111', '1', '0' and '0' respectively, the instruction is an unallocated memory hint.

Unallocated memory hints must be implemented as NOPs. Software must not use them, and they therefore have no UAL assembler syntax.

A6.7.55 LDRH (literal)

Load Register Halfword (literal) calculates an address from the PC value and an immediate offset, loads a halfword from memory, zero-extends it to form a 32-bit word, and writes it to a register. See *Memory accesses* on page A6-15 for information about memory accesses.

Encoding T1 ARMv7-M

```
LDRH<c> <Rt>,<label>
```

```
LDRH<c> <Rt>,[PC,#-0]
```

Special case

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	0	U	0	1	1	1	1	1	1	Rt				imm12											

```
if Rt == '1111' then SEE "Unallocated memory hints";
t = UInt(Rt);  imm32 = ZeroExtend(imm12, 32);  add = (U == '1');
if t == 13 then UNPREDICTABLE;
```

Assembler syntax

`LDRH<c><q> <Rt>, <label>`	Normal form
`LDRH<c><q> <Rt>, [PC, #+/-<imm>]`	Alternative form

where:

`<c><q>` See *Standard assembler syntax fields* on page A6-7.

`<Rt>` The destination register.

`<label>` The label of the literal data item that is to be loaded into `<Rt>`. The assembler calculates the required value of the offset from the PC value of the `ADR` instruction to this label. Permitted values of the offset are -4095 to 4095.

If the offset is zero or positive, `imm32` is equal to the offset and `add == TRUE`.

If the offset is negative, `imm32` is equal to minus the offset and `add == FALSE`.

The alternative syntax permits the addition or subtraction of the offset and the immediate offset to be specified separately, including permitting a subtraction of 0 that cannot be specified using the normal syntax. For more information, see *Use of labels in UAL instruction syntax* on page A4-5.

The pre-UAL syntax `LDR<c>H` is equivalent to `LDRH<c>`.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    base = Align(PC,4);
    address = if add then (base + imm32) else (base - imm32);
    data = MemU[address,2];
    R[t] = ZeroExtend(data, 32);
```

Exceptions

UsageFault, MemManage, BusFault.

A6.7.56 LDRH (register)

Load Register Halfword (register) calculates an address from a base register value and an offset register value, loads a halfword from memory, zero-extends it to form a 32-bit word, and writes it to a register. The offset register value can be shifted left by 0, 1, 2, or 3 bits. See *Memory accesses* on page A6-15 for information about memory accesses.

Encoding T1 All versions of the Thumb ISA.

`LDRH<c> <Rt>,[<Rn>,<Rm>]`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	0	1	Rm			Rn			Rt		

```
t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);
index = TRUE;  add = TRUE;  wback = FALSE;
(shift_t, shift_n) = (SRType_LSL, 0);
```

Encoding T2 ARMv7-M

`LDRH<c>.W <Rt>,[<Rn>,<Rm>{,LSL #<imm2>}]`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	0	0	0	1	1	Rn				Rt				0	0	0	0	0	0	imm2		Rm			

```
if Rn == '1111' then SEE LDRH (literal);
if Rt == '1111' then SEE "Unallocated memory hints";
t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);
index = TRUE;  add = TRUE;  wback = FALSE;
(shift_t, shift_n) = (SRType_LSL, UInt(imm2));
if t == 13 || m IN {13,15} then UNPREDICTABLE;
```

Assembler syntax

```
LDRH<c><q>  <Rt>, [<Rn>, <Rm> {, LSL #<shift>}]
```

where:

`<c><q>` See *Standard assembler syntax fields* on page A6-7.

`<Rt>` The destination register.

`<Rn>` Specifies the register that contains the base value. This register is allowed to be the SP.

`<Rm>` Contains the offset that is shifted left and added to the value of `<Rn>` to form the address.

`<shift>` Specifies the number of bits the value from `<Rm>` is shifted left, in the range 0-3. If this option is omitted, a shift by 0 is assumed and both encodings are permitted. If this option is specified, only encoding T2 is permitted.

The pre-UAL syntax `LDR<c>H` is equivalent to `LDRH<c>`.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    offset = Shift(R[m], shift_t, shift_n, APSR.C);
    offset_addr = if add then (R[n] + offset) else (R[n] - offset);
    address = if index then offset_addr else R[n];
    data = MemU[address,2];
    if wback then R[n] = offset_addr;
    R[t] = ZeroExtend(data, 32);
```

Exceptions

UsageFault, MemManage, BusFault.

A6.7.57 LDRHT

Load Register Halfword Unprivileged calculates an address from a base register value and an immediate offset, loads a halfword from memory, zero-extends it to form a 32-bit word, and writes it to a register. See *Memory accesses* on page A6-15 for information about memory accesses.

The memory access is restricted as if the processor were running unprivileged. (This makes no difference if the processor is actually running unprivileged.)

Encoding T1 ARMv7-M

`LDRHT<c> <Rt>,[<Rn>,#<imm8>]`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	0	0	0	1	1	Rn				Rt				1	1	1	0	imm8							

```
if Rn == '1111' then SEE LDRH (literal);
t = UInt(Rt);  n = UInt(Rn);  postindex = FALSE;  add = TRUE;
register_form = FALSE;  imm32 = ZeroExtend(imm8, 32);
if t IN {13,15} then UNPREDICTABLE;
```

Assembler syntax

```
LDRHT<c><q>  <Rt>, [<Rn> {, #<imm>}]
```

where:

`<c><q>` See *Standard assembler syntax fields* on page A6-7.

`<Rt>` Specifies the destination register.

`<Rn>` Specifies the base register. This register is allowed to be the SP.

`<imm>` Specifies the immediate offset added to the value of `<Rn>` to form the address. The range of allowed values is 0-255. `<imm>` can be omitted, meaning an offset of 0.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    address = R[n] + imm32;
    data = MemU_unpriv[address,2];
    R[t] = ZeroExtend(data, 32);
```

Exceptions

UsageFault, MemManage, BusFault.

A6.7.58 LDRSB (immediate)

Load Register Signed Byte (immediate) calculates an address from a base register value and an immediate offset, loads a byte from memory, sign-extends it to form a 32-bit word, and writes it to a register. It can use offset, post-indexed, or pre-indexed addressing. See *Memory accesses* on page A6-15 for information about memory accesses.

Encoding T1 ARMv7-M

```
LDRSB<c> <Rt>,[<Rn>,#<imm12>]
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	1	1	0	0	1	Rn				Rt				imm12											

```
if Rt == ’1111’ then SEE PLI;
if Rn == ’1111’ then SEE LDRSB (literal);
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm12, 32);
index = TRUE;  add = TRUE;  wback = FALSE;
if t == 13 then UNPREDICTABLE;
```

Encoding T2 ARMv7-M

```
LDRSB<c> <Rt>,[<Rn>,#-<imm8>]
LDRSB<c> <Rt>,[<Rn>],#+/-<imm8>
LDRSB<c> <Rt>,[<Rn>,#+/-<imm8>]!
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	1	0	0	0	1	Rn				Rt				1	P	U	W	imm8							

```
if Rt == '1111' && P == '1' && U == '0' && W == '0' then SEE PLI;
if Rn == '1111' then SEE LDRSB (literal);
if P == '1' && U == '1' && W == '0' then SEE LDRSBT;
if P == '0' && W == '0' then UNDEFINED;
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm8, 32);
index = (P == '1');  add = (U == '1');  wback = (W == '1');
if t IN {13,15} || (wback && n == t) then UNPREDICTABLE;
```

Assembler syntax

`LDRSB<c><q>  <Rt>, [<Rn> {, #+/-<imm>}]`	Offset: `index==TRUE`, `wback==FALSE`
`LDRSB<c><q>  <Rt>, [<Rn>, #+/-<imm>]!`	Pre-indexed: `index==TRUE`, `wback==TRUE`
`LDRSB<c><q>  <Rt>, [<Rn>], #+/-<imm>`	Post-indexed: `index==FALSE`, `wback==TRUE`

where:

`<c><q>` See *Standard assembler syntax fields* on page A6-7.

`<Rt>` Specifies the destination register.

`<Rn>` Specifies the base register. This register is allowed to be the SP. If this register is the PC, see *LDRSB (literal)* on page A6-120.

`+/-` Is + or omitted to indicate that the immediate offset is added to the base register value (`add == TRUE`), or – to indicate that the offset is to be subtracted (`add == FALSE`). Different instructions are generated for `#0` and `#-0`.

`<imm>` Specifies the immediate offset added to or subtracted from the value of `<Rn>` to form the address. The range of allowed values is 0-4095 for encoding T1, and 0-255 for encoding T2. For the offset addressing syntax, `<imm>` can be omitted, meaning an offset of 0.

The pre-UAL syntax `LDR<c>SB` is equivalent to `LDRSB<c>`.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    offset_addr = if add then (R[n] + imm32) else (R[n] - imm32);
    address = if index then offset_addr else R[n];
    R[t] = SignExtend(MemU[address,1], 32);
    if wback then R[n] = offset_addr;
```

Exceptions

MemManage, BusFault.

A6.7.59 LDRSB (literal)

Load Register Signed Byte (literal) calculates an address from the PC value and an immediate offset, loads a byte from memory, sign-extends it to form a 32-bit word, and writes it to a register. See *Memory accesses* on page A6-15 for information about memory accesses.

Encoding T1 ARMv7-M

`LDRSB<c> <Rt>,<label>`

`LDRSB<c> <Rt>,[PC,#-0]` Special case

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	1	U	0	0	1	1	1	1	1	Rt				imm12											

```
if Rt == '1111' then SEE PLI;
t = UInt(Rt);  imm32 = ZeroExtend(imm12, 32);  add = (U == '1');
if t == 13 then UNPREDICTABLE;
```

Assembler syntax

`LDRSB<c><q>  <Rt>, <label>`	Normal form
`LDRSB<c><q>  <Rt>, [PC, #+/-<imm>]`	Alternative form

where:

`<c><q>` See *Standard assembler syntax fields* on page A6-7.

`<Rt>` The destination register.

`<label>` The label of the literal data item that is to be loaded into `<Rt>`. The assembler calculates the required value of the offset from the PC value of the `ADR` instruction to this label. Permitted values of the offset are -4095 to 4095.

If the offset is zero or positive, `imm32` is equal to the offset and `add == TRUE`.

If the offset is negative, `imm32` is equal to minus the offset and `add == FALSE`.

The alternative syntax permits the addition or subtraction of the offset and the immediate offset to be specified separately, including permitting a subtraction of 0 that cannot be specified using the normal syntax. For more information, see *Use of labels in UAL instruction syntax* on page A4-5.

The pre-UAL syntax `LDR<c>SB` is equivalent to `LDRSB<c>`.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    base = Align(PC,4);
    address = if add then (base + imm32) else (base - imm32);
    R[t] = SignExtend(MemU[address,1], 32);
```

Exceptions

MemManage, BusFault.

A6.7.60 LDRSB (register)

Load Register Signed Byte (register) calculates an address from a base register value and an offset register value, loads a byte from memory, sign-extends it to form a 32-bit word, and writes it to a register. The offset register value can be shifted left by 0, 1, 2, or 3 bits. See *Memory accesses* on page A6-15 for information about memory accesses.

Encoding T1 All versions of the Thumb ISA.

```
LDRSB<c> <Rt>,[<Rn>,<Rm>]
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	0	1	1	Rm			Rn			Rt		

```
t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);
index = TRUE;  add = TRUE;  wback = FALSE;
(shift_t, shift_n) = (SRType_LSL, 0);
```

Encoding T2 ARMv7-M

```
LDRSB<c>.W <Rt>,[<Rn>,<Rm>{,LSL #<imm2>}]
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	1	0	0	0	1	Rn				Rt				0	0	0	0	0	0	imm2		Rm			

```
if Rt == '1111' then SEE PLI;
if Rn == '1111' then SEE LDRSB (literal);
t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);
index = TRUE;  add = TRUE;  wback = FALSE;
(shift_t, shift_n) = (SRType_LSL, UInt(imm2));
if t == 13 || m IN {13,15} then UNPREDICTABLE;
```

Assembler syntax

```
LDRSB<c><q>  <Rt>, [<Rn>, <Rm> {, LSL #<shift>}]
```

where:

`<c><q>` See *Standard assembler syntax fields* on page A6-7.

`<Rt>` The destination register.

`<Rn>` Specifies the register that contains the base value. This register is allowed to be the SP.

`<Rm>` Contains the offset that is shifted left and added to the value of `<Rn>` to form the address.

`<shift>` Specifies the number of bits the value from `<Rm>` is shifted left, in the range 0-3. If this option is omitted, a shift by 0 is assumed and both encodings are permitted. If this option is specified, only encoding T2 is permitted.

The pre-UAL syntax `LDR<c>SB` is equivalent to `LDRSB<c>`.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    offset = Shift(R[m], shift_t, shift_n, APSR.C);
    offset_addr = if add then (R[n] + offset) else (R[n] - offset);
    address = if index then offset_addr else R[n];
    R[t] = SignExtend(MemU[address,1], 32);
```

Exceptions

MemManage, BusFault.

A6.7.61 LDRSBT

Load Register Signed Byte Unprivileged calculates an address from a base register value and an immediate offset, loads a byte from memory, sign-extends it to form a 32-bit word, and writes it to a register. See *Memory accesses* on page A6-15 for information about memory accesses.

The memory access is restricted as if the processor were running unprivileged. (This makes no difference if the processor is actually running unprivileged.)

Encoding T1 ARMv7-M

```
LDRSBT<c> <Rt>,[<Rn>,#<imm8>]
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	1	0	0	0	1	Rn				Rt				1	1	1	0	imm8							

```
if Rn == '1111' then SEE LDRSB (literal);
t = UInt(Rt);  n = UInt(Rn);  postindex = FALSE;  add = TRUE;
register_form = FALSE;  imm32 = ZeroExtend(imm8, 32);
if t IN {13,15} then UNPREDICTABLE;
```

Assembler syntax

```
LDRSBT<c><q>  <Rt>, [<Rn> {, #<imm>}]
```

where:

`<c><q>` See *Standard assembler syntax fields* on page A6-7.

`<Rt>` Specifies the destination register.

`<Rn>` Specifies the base register. This register is allowed to be the SP.

`<imm>` Specifies the immediate offset added to the value of `<Rn>` to form the address. The range of allowed values is 0-255. `<imm>` can be omitted, meaning an offset of 0.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    address = R[n] + imm32;
    R[t] = SignExtend(MemU_unpriv[address,1], 32);
```

Exceptions

MemManage, BusFault.

A6.7.62 LDRSH (immediate)

Load Register Signed Halfword (immediate) calculates an address from a base register value and an immediate offset, loads a halfword from memory, sign-extends it to form a 32-bit word, and writes it to a register. It can use offset, post-indexed, or pre-indexed addressing. See *Memory accesses* on page A6-15 for information about memory accesses.

Encoding T1 ARMv7-M

```
LDRSH<c> <Rt>,[<Rn>,#<imm12>]
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	1	1	0	1	1	Rn				Rt				imm12											

```
if Rn == '1111' then SEE LDRSH (literal);
if Rt == '1111' then SEE "Unallocated memory hints";
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm12, 32);
index = TRUE;  add = TRUE;  wback = FALSE;
if t == 13 then UNPREDICTABLE;
```

Encoding T2 ARMv7-M

```
LDRSH<c> <Rt>,[<Rn>,#-<imm8>]
LDRSH<c> <Rt>,[<Rn>],#+/-<imm8>
LDRSH<c> <Rt>,[<Rn>,#+/-<imm8>]!
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	1	0	0	1	1	Rn				Rt				1	P	U	W	imm8							

```
if Rn == '1111' then SEE LDRSH (literal);
if Rt == '1111' && P == '1' && U == '0' && W == '0' then SEE "Unallocated memory hints";
if P == '1' && U == '1' && W == '0' then SEE LDRSHT;
if P == '0' && W == '0' then UNDEFINED;
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm8, 32);
index = (P == '1');  add = (U == '1');  wback = (W == '1');
if t IN {13,15} || (wback && n == t) then UNPREDICTABLE;
```

Assembler syntax

`LDRSH<c><q>  <Rt>, [<Rn> {, #+/-<imm>}]`	Offset: `index==TRUE`, `wback==FALSE`
`LDRSH<c><q>  <Rt>, [<Rn>, #+/-<imm>]!`	Pre-indexed: `index==TRUE`, `wback==TRUE`
`LDRSH<c><q>  <Rt>, [<Rn>], #+/-<imm>`	Post-indexed: `index==FALSE`, `wback==TRUE`

where:

`<c><q>` See *Standard assembler syntax fields* on page A6-7.

`<Rt>` Specifies the destination register.

`<Rn>` Specifies the base register. This register is allowed to be the SP. If this register is the PC, see *LDRSH (literal)* on page A6-128.

`+/-` Is + or omitted to indicate that the immediate offset is added to the base register value (`add == TRUE`), or – to indicate that the offset is to be subtracted (`add == FALSE`). Different instructions are generated for `#0` and `#-0`.

`<imm>` Specifies the immediate offset added to or subtracted from the value of `<Rn>` to form the address. The range of allowed values is 0-4095 for encoding T1, and 0-255 for encoding T2. For the offset addressing syntax, `<imm>` can be omitted, meaning an offset of 0.

The pre-UAL syntax `LDR<c>SH` is equivalent to `LDRSH<c>`.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    offset_addr = if add then (R[n] + imm32) else (R[n] - imm32);
    address = if index then offset_addr else R[n];
    data = MemU[address,2];
    if wback then R[n] = offset_addr;
    R[t] = SignExtend(data, 32);
```

Exceptions

UsageFault, MemManage, BusFault.

A6.7.63 LDRSH (literal)

Load Register Signed Halfword (literal) calculates an address from the PC value and an immediate offset, loads a halfword from memory, sign-extends it to form a 32-bit word, and writes it to a register. See *Memory accesses* on page A6-15 for information about memory accesses.

Encoding T1 ARMv7-M

`LDRSH<c> <Rt>,<label>`

`LDRSH<c> <Rt>,[PC,#-0]` Special case

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	1	U	0	1	1	1	1	1	1	Rt				imm12											

```
if Rt == '1111' then SEE "Unallocated memory hints";
t = UInt(Rt);  imm32 = ZeroExtend(imm12, 32);  add = (U == '1');
if t == 13 then UNPREDICTABLE;
```

Assembler syntax

`LDRSH<c><q>  <Rt>, <label>`	Normal form
`LDRSH<c><q>  <Rt>, [PC, #+/-<imm>]`	Alternative form

where:

`<c><q>` See *Standard assembler syntax fields* on page A6-7.

`<Rt>` The destination register.

`<label>` The label of the literal data item that is to be loaded into `<Rt>`. The assembler calculates the required value of the offset from the `PC` value of the `ADR` instruction to this label. Permitted values of the offset are -4095 to 4095.

If the offset is zero or positive, `imm32` is equal to the offset and `add == TRUE`.

If the offset is negative, `imm32` is equal to minus the offset and `add == FALSE`.

The alternative syntax permits the addition or subtraction of the offset and the immediate offset to be specified separately, including permitting a subtraction of 0 that cannot be specified using the normal syntax. For more information, see *Use of labels in UAL instruction syntax* on page A4-5.

The pre-UAL syntax `LDR<c>SH` is equivalent to `LDRSH<c>`.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    base = Align(PC,4);
    address = if add then (base + imm32) else (base - imm32);
    data = MemU[address,2];
    R[t] = SignExtend(data, 32);
```

Exceptions

UsageFault, MemManage, BusFault.

Unallocated memory hints

If the Rt field is '1111' in encoding T1, the instruction is an unallocated memory hint.

Unallocated memory hints must be implemented as NOPs. Software must not use them, and they therefore have no UAL assembler syntax.

A6.7.64 LDRSH (register)

Load Register Signed Halfword (register) calculates an address from a base register value and an offset register value, loads a halfword from memory, sign-extends it to form a 32-bit word, and writes it to a register. The offset register value can be shifted left by 0, 1, 2, or 3 bits. See *Memory accesses* on page A6-15 for information about memory accesses.

Encoding T1

All versions of the Thumb ISA.

```
LDRSH<c> <Rt>,[<Rn>,<Rm>]
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	Rm			Rn			Rt		

```
t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);
index = TRUE;  add = TRUE;  wback = FALSE;
(shift_t, shift_n) = (SRType_LSL, 0);
```

Encoding T2

ARMv7-M

```
LDRSH<c>.W <Rt>,[<Rn>,<Rm>{,LSL #<imm2>}]
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	1	0	0	1	1	Rn				Rt				0	0	0	0	0	0	imm2		Rm			

```
if Rn == '1111' then SEE LDRSH (literal);
if Rt == '1111' then SEE "Unallocated memory hints";
t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);
index = TRUE;  add = TRUE;  wback = FALSE;
(shift_t, shift_n) = (SRType_LSL, UInt(imm2));
if t == 13 || m IN {13,15} then UNPREDICTABLE;
```

Assembler syntax

```
LDRSH<c><q>  <Rt>, [<Rn>, <Rm> {, LSL #<shift>}]
```

where:

`<c><q>` See *Standard assembler syntax fields* on page A6-7.

`<Rt>` Specifies the destination register.

`<Rn>` Specifies the register that contains the base value. This register is allowed to be the SP.

`<Rm>` Contains the offset that is shifted left and added to the value of `<Rn>` to form the address.

`<shift>` Specifies the number of bits the value from `<Rm>` is shifted left, in the range 0-3. If this option is omitted, a shift by 0 is assumed and both encodings are permitted. If this option is specified, only encoding T2 is permitted.

The pre-UAL syntax `LDR<c>SH` is equivalent to `LDRSH<c>`.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    offset = Shift(R[m], shift_t, shift_n, APSR.C);
    offset_addr = if add then (R[n] + offset) else (R[n] - offset);
    address = if index then offset_addr else R[n];
    data = MemU[address,2];
    if wback then R[n] = offset_addr;
    R[t] = SignExtend(data, 32);
```

Exceptions

UsageFault, MemManage, BusFault.

A6.7.65 LDRSHT

Load Register Signed Halfword Unprivileged calculates an address from a base register value and an immediate offset, loads a halfword from memory, sign-extends it to form a 32-bit word, and writes it to a register. See *Memory accesses* on page A6-15 for information about memory accesses.

The memory access is restricted as if the processor were running unprivileged. (This makes no difference if the processor is actually running unprivileged.)

Encoding T1 ARMv7-M

`LDRSHT<c> <Rt>,[<Rn>,#<imm8>]`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	1	0	0	1	1	Rn				Rt				1	1	1	0	imm8							

```
if Rn == '1111' then SEE LDRSH (literal);
t = UInt(Rt);  n = UInt(Rn);  postindex = FALSE;  add = TRUE;
register_form = FALSE;  imm32 = ZeroExtend(imm8, 32);
if t IN {13,15} then UNPREDICTABLE;
```

Assembler syntax

`LDRSHT<c><q>  <Rt>, [<Rn>, {, #<imm>}]`

where:

`<c><q>` See *Standard assembler syntax fields* on page A6-7.

`<Rt>` Specifies the destination register.

`<Rn>` Specifies the base register. This register is allowed to be the SP.

`<imm>` Specifies the immediate offset added to the value of `<Rm>` to form the address. The range of allowed values is 0-255. <imm> can be omitted, meaning an offset of 0.

The pre-UAL syntax `LDR<c>SH` is equivalent to `LDRSH<c>`.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    address = R[n] + imm32;
    data = MemU_unpriv[address,2];
    R[t] = SignExtend(data, 32);
```

Exceptions

UsageFault, MemManage, BusFault.

A6.7.66 LDRT

Load Register Unprivileged calculates an address from a base register value and an immediate offset, loads a word from memory, and writes it to a register. See *Memory accesses* on page A6-15 for information about memory accesses.

The memory access is restricted as if the processor were running unprivileged. (This makes no difference if the processor is actually running unprivileged.)

Encoding T1 ARMv7-M

`LDRT<c> <Rt>,[<Rn>,#<imm8>]`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	0	0	1	0	1	Rn				Rt				1	1	1	0	imm8							

```
if Rn == '1111' then SEE LDR (literal);
t = UInt(Rt);  n = UInt(Rn);  postindex = FALSE;  add = TRUE;
register_form = FALSE;  imm32 = ZeroExtend(imm8, 32);
if t IN {13,15} then UNPREDICTABLE;
```

Assembler syntax

`LDRT<c><q>  <Rt>, [<Rn> {, #<imm>}]`

where:

`<c><q>` See *Standard assembler syntax fields* on page A6-7.

`<Rt>` Specifies the destination register.

`<Rn>` Specifies the base register. This register is allowed to be the SP.

`<imm>` Specifies the immediate offset added to the value of `<Rn>` to form the address. The range of allowed values is 0-255. `<imm>` can be omitted, meaning an offset of 0.

The pre-UAL syntax `LDR<c>T` is equivalent to `LDRT<c>`.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    address = R[n] + imm32;
    data = MemU_unpriv[address,4];
    R[t] = data;
```

Exceptions

UsageFault, MemManage, BusFault.

A6.7.67 LSL (immediate)

Logical Shift Left (immediate) shifts a register value left by an immediate number of bits, shifting in zeros, and writes the result to the destination register. It can optionally update the condition flags based on the result.

Encoding T1 All versions of the Thumb ISA.

`LSLS <Rd>,<Rm>,#<imm5>`	Outside IT block.
`LSL<c> <Rd>,<Rm>,#<imm5>`	Inside IT block.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	imm5					Rm			Rd		

```
if imm5 == '00000' then SEE MOV (register);
d = UInt(Rd);  m = UInt(Rm);  setflags = !InITBlock();
(-, shift_n) = DecodeImmShift('00', imm5);
```

Encoding T2 ARMv7-M

`LSL{S}<c>.W <Rd>,<Rm>,#<imm5>`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	0	1	0	0	1	0	S	1	1	1	1	(0)	imm3			Rd				imm2		0	0	Rm			

```
if (imm3:imm2) == '00000' then SEE MOV (register);
d = UInt(Rd);  m = UInt(Rm);  setflags = (S == '1');
(-, shift_n) = DecodeImmShift('00', imm3:imm2);
if d IN {13,15} || m IN {13,15} then UNPREDICTABLE;
```

Assembler syntax

```
LSL{S}<c><q>  <Rd>, <Rm>, #<imm5>
```

where:

`S` If present, specifies that the instruction updates the flags. Otherwise, the instruction does not update the flags.

`<c><q>` See *Standard assembler syntax fields* on page A6-7.

`<Rd>` Specifies the destination register.

`<Rm>` Specifies the register that contains the first operand.

`<imm5>` Specifies the shift amount, in the range 0 to 31. See *Shifts applied to a register* on page A6-12.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    (result, carry) = Shift_C(R[m], SRType_LSL, shift_n, APSR.C);
    R[d] = result;
    if setflags then
        APSR.N = result<31>;
        APSR.Z = IsZeroBit(result);
        APSR.C = carry;
        // APSR.V unchanged
```

Exceptions

None.

A6.7.68 LSL (register)

Logical Shift Left (register) shifts a register value left by a variable number of bits, shifting in zeros, and writes the result to the destination register. The variable number of bits is read from the bottom byte of a register. It can optionally update the condition flags based on the result.

Encoding T1

All versions of the Thumb ISA.

```
LSLS    <Rdn>,<Rm>                                      Outside IT block.
LSL<c> <Rdn>,<Rm>                                       Inside IT block.
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	0	0	0	1	0	Rm			Rdn		

```
d = UInt(Rdn);  n = UInt(Rdn);  m = UInt(Rm);  setflags = !InITBlock();
```

Encoding T2

ARMv7-M

```
LSL{S}<c>.W <Rd>,<Rn>,<Rm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	1	0	0	0	0	S	Rn				1	1	1	1	Rd				0	0	0	0	Rm			

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  setflags = (S == '1');
if d IN {13,15} || n IN {13,15} || m IN {13,15} then UNPREDICTABLE;
```

Assembler syntax

```
LSL{S}<c><q>  <Rd>, <Rn>, <Rm>
```

where:

S   If present, specifies that the instruction updates the flags. Otherwise, the instruction does not update the flags.

<c><q>   See *Standard assembler syntax fields* on page A6-7.

<Rd>   Specifies the destination register.

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

<Rm>   Specifies the register whose bottom byte contains the amount to shift by.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    shift_n = UInt(R[m]<7:0>);
    (result, carry) = Shift_C(R[n], SRType_LSL, shift_n, APSR.C);
    R[d] = result;
    if setflags then
        APSR.N = result<31>;
        APSR.Z = IsZeroBit(result);
        APSR.C = carry;
        // APSR.V unchanged
```

Exceptions

None.

A6.7.69 LSR (immediate)

Logical Shift Right (immediate) shifts a register value right by an immediate number of bits, shifting in zeros, and writes the result to the destination register. It can optionally update the condition flags based on the result.

Encoding T1 All versions of the Thumb ISA.

`LSRS <Rd>,<Rm>,#<imm5>`	Outside IT block.
`LSR<c> <Rd>,<Rm>,#<imm5>`	Inside IT block.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	imm5					Rm			Rd		

```
d = UInt(Rd);  m = UInt(Rm);  setflags = !InITBlock();
(-, shift_n) = DecodeImmShift('01', imm5);
```

Encoding T2 ARMv7-M

`LSR{S}<c>.W <Rd>,<Rm>,#<imm5>`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	0	1	0	0	1	0	S	1	1	1	1	(0)	imm3			Rd				imm2		0	1	Rm			

```
d = UInt(Rd);  m = UInt(Rm);  setflags = (S == '1');
(-, shift_n) = DecodeImmShift('01', imm3:imm2);
if d IN {13,15} || m IN {13,15} then UNPREDICTABLE;
```

Assembler syntax

```
LSR{S}<c><q>  <Rd>, <Rm>, #<imm5>
```

where:

`S` If present, specifies that the instruction updates the flags. Otherwise, the instruction does not update the flags.

`<c><q>` See *Standard assembler syntax fields* on page A6-7.

`<Rd>` Specifies the destination register.

`<Rm>` Specifies the register that contains the first operand.

`<imm5>` Specifies the shift amount, in the range 1 to 32. See *Shifts applied to a register* on page A6-12.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    (result, carry) = Shift_C(R[m], SRType_LSR, shift_n, APSR.C);
    R[d] = result;
    if setflags then
        APSR.N = result<31>;
        APSR.Z = IsZeroBit(result);
        APSR.C = carry;
        // APSR.V unchanged
```

Exceptions

None.

A6.7.70 LSR (register)

Logical Shift Right (register) shifts a register value right by a variable number of bits, shifting in zeros, and writes the result to the destination register. The variable number of bits is read from the bottom byte of a register. It can optionally update the condition flags based on the result.

Encoding T1 All versions of the Thumb ISA.

`LSRS <Rdn>,<Rm>`	Outside IT block.
`LSR<c> <Rdn>,<Rm>`	Inside IT block.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	0	0	0	1	1	Rm			Rdn		

```
d = UInt(Rdn);  n = UInt(Rdn);  m = UInt(Rm);  setflags = !InITBlock();
```

Encoding T2 ARMv7-M

`LSR{S}<c>.W <Rd>,<Rn>,<Rm>`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	1	0	0	0	1	S	Rn				1	1	1	1	Rd				0	0	0	0	Rm			

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  setflags = (S == '1');
if d IN {13,15} || n IN {13,15} || m IN {13,15} then UNPREDICTABLE;
```

Assembler syntax

```
LSR{S}<c><q>  <Rd>, <Rn>, <Rm>
```

where:

`S` If present, specifies that the instruction updates the flags. Otherwise, the instruction does not update the flags.

`<c><q>` See *Standard assembler syntax fields* on page A6-7.

`<Rd>` Specifies the destination register.

`<Rn>` Specifies the register that contains the first operand.

`<Rm>` Specifies the register whose bottom byte contains the amount to shift by.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    shift_n = UInt(R[m]<7:0>);
    (result, carry) = Shift_C(R[n], SRType_LSR, shift_n, APSR.C);
    R[d] = result;
    if setflags then
        APSR.N = result<31>;
        APSR.Z = IsZeroBit(result);
        APSR.C = carry;
        // APSR.V unchanged
```

Exceptions

None.

A6.7.71 MCR, MCR2

Move to Coprocessor from ARM Register passes the value of an ARM register to a coprocessor.

If no coprocessor can execute the instruction, a UsageFault exception is generated.

Encoding T1 ARMv7-M

`MCR<c> <coproc>,<opc1>,<Rt>,<CRn>,<CRm>{,<opc2>}`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	1	0	opc1			0	CRn				Rt				coproc				opc2			1	CRm			

```
t = UInt(Rt);  cp = UInt(coproc);
if t == 15 || t == 13 then UNPREDICTABLE;
```

Encoding T2 ARMv7-M

`MCR2<c> <coproc>,<opc1>,<Rt>,<CRn>,<CRm>{,<opc2>}`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	0	opc1			0	CRn				Rt				coproc				opc2			1	CRm			

```
t = UInt(Rt);  cp = UInt(coproc);
if t == 15 || t == 13 then UNPREDICTABLE;
```

Assembler syntax

```
MCR{2}<c><q>  <coproc>, #<opc1>, <Rt>, <CRn>, <CRm>{, #<opc2>}
```

where:

`2` If specified, selects the C == 1 form of the encoding. If omitted, selects the C == 0 form.

`<c><q>` See *Standard assembler syntax fields* on page A6-7.

`<coproc>` Specifies the name of the coprocessor. The standard generic coprocessor names are p0, p1, ..., p15.

`<opc1>` Is a coprocessor-specific opcode in the range 0 to 7.

`<Rt>` Is the ARM register whose value is transferred to the coprocessor.

`<CRn>` Is the destination coprocessor register.

`<CRm>` Is an additional destination coprocessor register.

`<opc2>` Is a coprocessor-specific opcode in the range 0-7. If it is omitted, `<opc2>` is assumed to be 0.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    if !Coproc_Accepted(cp, ThisInstr()) then
        GenerateCoprocessorException();
    else
        Coproc_SendOneWord(R[t], cp, ThisInstr());
```

Exceptions

UsageFault.

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.

A6.7.72 MCRR, MCRR2

Move to Coprocessor from two ARM Registers passes the values of two ARM registers to a coprocessor.

If no coprocessor can execute the instruction, a UsageFault exception is generated.

Encoding T1 ARMv7-M

`MCRR<c> <coproc>,<opc1>,<Rt>,<Rt2>,<CRm>`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	0	0	0	1	0	0	Rt2				Rt				coproc				opc1				CRm			

```
t = UInt(Rt);  t2 = UInt(Rt2);  cp = UInt(coproc);
if t == 15 || t2 == 15 then UNPREDICTABLE;
if t == 13 || t2 == 13 then UNPREDICTABLE;
```

Encoding T2 ARMv7-M

`MCRR2<c> <coproc>,<opc1>,<Rt>,<Rt2>,<CRm>`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	0	0	0	1	0	0	Rt2				Rt				coproc				opc1				CRm			

```
t = UInt(Rt);  t2 = UInt(Rt2);  cp = UInt(coproc);
if t == 15 || t2 == 15 then UNPREDICTABLE;
if t == 13 || t2 == 13 then UNPREDICTABLE;
```

Assembler syntax

```
MCRR{2}<c><q>  <coproc>, #<opc1>, <Rt>, <Rt2>, <CRm>
```

where:

`2`	If specified, selects the C ==1 form of the encoding. If omitted, selects the C == 0 form.
`<c><q>`	See *Standard assembler syntax fields* on page A6-7.
`<coproc>`	Specifies the name of the coprocessor. The standard generic coprocessor names are p0, p1, ..., p15.
`<opc1>`	Is a coprocessor-specific opcode in the range 0 to 15.
`<Rt>`	Is the first ARM register whose value is transferred to the coprocessor.
`<Rt2>`	Is the second ARM register whose value is transferred to the coprocessor.
`<CRm>`	Is the destination coprocessor register.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    if !Coproc_Accepted(cp, ThisInstr()) then
        GenerateCoprocessorException();
    else
        Coproc_SendTwoWords(R[t], R[t2], cp, ThisInstr());
```

Exceptions

UsageFault.

A6.7.73 MLA

Multiply Accumulate multiplies two register values, and adds a third register value. The least significant 32 bits of the result are written to the destination register. These 32 bits do not depend on whether signed or unsigned calculations are performed.

Encoding T1 ARMv7-M

```
MLA<c> <Rd>,<Rn>,<Rm>,<Ra>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	1	1	0	0	0	0	Rn				Ra				Rd				0	0	0	0	Rm			

```
if Ra == '1111' then SEE MUL;
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  a = UInt(Ra);  setflags = FALSE;
if d IN {13,15} || n IN {13,15} || m IN {13,15} || a == 13 then UNPREDICTABLE;
```

Assembler syntax

```
MLA<c><q>  <Rd>, <Rn>, <Rm>, <Ra>
```

where:

`<c><q>` See *Standard assembler syntax fields* on page A6-7.

`<Rd>` Specifies the destination register.

`<Rn>` Specifies the register that contains the first operand.

`<Rm>` Specifies the register that contains the second operand.

`<Ra>` Specifies the register containing the accumulate value.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    operand1 = SInt(R[n]);  // operand1 = UInt(R[n]) produces the same final results
    operand2 = SInt(R[m]);  // operand2 = UInt(R[m]) produces the same final results
    addend   = SInt(R[a]);  // addend   = UInt(R[a]) produces the same final results
    result = operand1 * operand2 + addend;
    R[d] = result<31:0>;
    if setflags then
        APSR.N = result<31>;
        APSR.Z = IsZeroBit(result);
        // APSR.C unchanged
        // APSR.V unchanged
```

Exceptions

None.

A6.7.74 MLS

Multiply and Subtract multiplies two register values, and subtracts the least significant 32 bits of the result from a third register value. These 32 bits do not depend on whether signed or unsigned calculations are performed. The result is written to the destination register.

Encoding T1 ARMv7-M

`MLS<c> <Rd>,<Rn>,<Rm>,<Ra>`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	1	1	0	0	0	0	Rn				Ra				Rd				0	0	0	1	Rm			

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  a = UInt(Ra);
if d IN {13,15} || n IN {13,15} || m IN {13,15} || a IN {13,15} then UNPREDICTABLE;
```

Assembler syntax

```
MLS<c><q>  <Rd>, <Rn>, <Rm>, <Ra>
```

where:

`<c><q>` See *Standard assembler syntax fields* on page A6-7.

`<Rd>` Specifies the destination register.

`<Rn>` Specifies the register that contains the first operand.

`<Rm>` Specifies the register that contains the second operand.

`<Ra>` Specifies the register containing the accumulate value.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    operand1 = SInt(R[n]);  // operand1 = UInt(R[n]) produces the same final results
    operand2 = SInt(R[m]);  // operand2 = UInt(R[m]) produces the same final results
    addend   = SInt(R[a]);  // addend   = UInt(R[a]) produces the same final results
    result = addend - operand1 * operand2;
    R[d] = result<31:0>;
```

Exceptions

None.

A6.7.75 MOV (immediate)

Move (immediate) writes an immediate value to the destination register. It can optionally update the condition flags based on the value.

Encoding T1

All versions of the Thumb ISA.

```
MOVS <Rd>,#<imm8>                                   Outside IT block.
MOV<c> <Rd>,#<imm8>                                 Inside IT block.
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	0	Rd			imm8							

```
d = UInt(Rd);  setflags = !InITBlock();  imm32 = ZeroExtend(imm8, 32);  carry = APSR.C;
```

Encoding T2

ARMv7-M

```
MOV{S}<c>.W <Rd>,#<const>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	i	0	0	0	1	0	S	1	1	1	1	0	imm3			Rd				imm8							

```
d = UInt(Rd);  setflags = (S == '1');  (imm32, carry) = ThumbExpandImm_C(i:imm3:imm8, APSR.C);
if d IN {13,15} then UNPREDICTABLE;
```

Encoding T3

ARMv7-M

```
MOVW<c> <Rd>,#<imm16>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	i	1	0	0	1	0	0	imm4				0	imm3			Rd				imm8							

```
d = UInt(Rd);  setflags = FALSE;  imm32 = ZeroExtend(imm4:i:imm3:imm8, 32);
if d IN {13,15} then UNPREDICTABLE;
```

Assembler syntax

`MOV{S}<c><q>  <Rd>, #<const>`	All encodings permitted
`MOVW<c><q>  <Rd>, #<const>`	Only encoding T3 permitted

where:

`S` If present, specifies that the instruction updates the flags. Otherwise, the instruction does not update the flags.

`<c><q>` See *Standard assembler syntax fields* on page A6-7.

`<Rd>` Specifies the destination register.

`<const>` Specifies the immediate value to be placed in `<Rd>`. The range of allowed values is 0-255 for encoding T1 and 0-65535 for encoding T3. See *Modified immediate constants in Thumb instructions* on page A5-15 for the range of allowed values for encoding T2.

When both 32-bit encodings are available for an instruction, encoding T2 is preferred to encoding T3 (if encoding T3 is required, use the `MOVW` syntax).

The pre-UAL syntax `MOV<c>S` is equivalent to `MOVS<c>`.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    result = imm32;
    R[d] = result;
    if setflags then
        APSR.N = result<31>;
        APSR.Z = IsZeroBit(result);
        APSR.C = carry;
        // APSR.V unchanged
```

Exceptions

None.

A6.7.76 MOV (register)

Move (register) copies a value from a register to the destination register. It can optionally update the condition flags based on the value.

Encoding T1 ARMv6-M, ARMv7-M

If `<Rd>` and `<Rm>` both from R0-R7,
otherwise all versions of the Thumb ISA.
If `<Rd>` is the PC, must be outside or last in IT block

```
MOV<c> <Rd>,<Rm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	1	1	0	D	Rm				Rd		

```
d = UInt(D:Rd);  m = UInt(Rm);  setflags = FALSE;
if d == 15 && InITBlock() && !LastInITBlock() then UNPREDICTABLE;
```

Encoding T2 All versions of the Thumb ISA.

```
MOVS <Rd>,<Rm>
```

~~(formerly LSL <Rd>,<Rm>,#0)~~ Not permitted inside IT block

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	0	Rm			Rd		

```
d = UInt(Rd);  m = UInt(Rm);  setflags = TRUE;
if InITBlock() then UNPREDICTABLE;
```

Encoding T3 ARMv7-M

```
MOV{S}<c>.W <Rd>,<Rm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	0	1	0	0	1	0	S	1	1	1	1	(0)	0	0	0	Rd				0	0	0	0	Rm			

```
d = UInt(Rd);  m = UInt(Rm);  setflags = (S == '1');
if setflags && (d IN {13,15} || m IN {13,15}) then UNPREDICTABLE;
if !setflags && (d == 15 || m == 15 || (d == 13 && m == 13)) then UNPREDICTABLE;
```

Assembler syntax

```
MOV{S}<c><q>  <Rd>, <Rm>
```

where:

`S` If present, specifies that the instruction updates the flags. Otherwise, the instruction does not update the flags.

`<c><q>` See *Standard assembler syntax fields* on page A6-7.

`<Rd>` The destination register. This register can be the SP or PC, provided S is not specified.

If `<Rd>` is the PC, then only encoding T1 is permitted, and the instruction causes a branch to the address moved to the PC. The instruction must either be outside an IT block or the last instruction of an IT block.

`<Rm>` The source register. This register can be the SP or PC.The instruction must not specify `S` if `<Rm>` is the SP or PC.

Encoding T3 is not permitted if either:

- `<Rd>` or `<Rm>` is the PC
- both `<Rd>` and `<Rm>` are the SP.

Note

ARM deprecates the use of the following `MOV` (register) instructions:

- ones in which `<Rd>` is the SP or PC and `<Rm>` is also the SP or PC is deprecated.
- ones in which S is specified and `<Rm>` is the SP, or `<Rm>` is the PC.

The pre-UAL syntax `MOV<c>S` is equivalent to `MOVS<c>`.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    result = R[m];
    if d == 15 then
        ALUWritePC(result);  // setflags is always FALSE here
    else
        R[d] = result;
        if setflags then
            APSR.N = result<31>;
            APSR.Z = IsZeroBit(result);
            // APSR.C unchanged
            // APSR.V unchanged
```

Exceptions

None.

A6.7.77 MOV (shifted register)

Move (shifted register) is a synonym for ASR, LSL, LSR, ROR, and RRX.

See the following sections for details:

- *ASR (immediate)* on page A6-36
- *ASR (register)* on page A6-38
- *LSL (immediate)* on page A6-134
- *LSL (register)* on page A6-136
- *LSR (immediate)* on page A6-138
- *LSR (register)* on page A6-140
- *ROR (immediate)* on page A6-194
- *ROR (register)* on page A6-196
- *RRX* on page A6-198.

Assembler syntax

Table A6-4 shows the equivalences between MOV (shifted register) and other instructions.

Table A6-4 MOV (shift, register shift) equivalences)

MOV instruction	Canonical form
`MOV{S} <Rd>,<Rm>,ASR #<n>`	`ASR{S} <Rd>,<Rm>,#<n>`
`MOV{S} <Rd>,<Rm>,LSL #<n>`	`LSL{S} <Rd>,<Rm>,#<n>`
`MOV{S} <Rd>,<Rm>,LSR #<n>`	`LSR{S} <Rd>,<Rm>,#<n>`
`MOV{S} <Rd>,<Rm>,ROR #<n>`	`ROR{S} <Rd>,<Rm>,#<n>`
`MOV{S} <Rd>,<Rm>,ASR <Rs>`	`ASR{S} <Rd>,<Rm>,<Rs>`
`MOV{S} <Rd>,<Rm>,LSL <Rs>`	`LSL{S} <Rd>,<Rm>,<Rs>`
`MOV{S} <Rd>,<Rm>,LSR <Rs>`	`LSR{S} <Rd>,<Rm>,<Rs>`
`MOV{S} <Rd>,<Rm>,ROR <Rs>`	`ROR{S} <Rd>,<Rm>,<Rs>`
`MOV{S} <Rd>,<Rm>,RRX`	`RRX{S} <Rd>,<Rm>`

The canonical form of the instruction is produced on disassembly.

Exceptions

None.

A6.7.78 MOVT

Move Top writes an immediate value to the top halfword of the destination register. It does not affect the contents of the bottom halfword.

Encoding T1 ARMv7-M

```
MOVT<c> <Rd>,#<imm16>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	i	1	0	1	1	0	0	imm4				0	imm3			Rd				imm8							

```
d = UInt(Rd);  imm16 = imm4:i:imm3:imm8;
if d IN {13,15} then UNPREDICTABLE;
```

Assembler syntax

```
MOVT<c><q>  <Rd>, #<imm16>
```

where:

`<c><q>` See *Standard assembler syntax fields* on page A6-7.

`<Rd>` Specifies the destination register.

`<imm16>` Specifies the immediate value to be written to `<Rd>`. It must be in the range 0-65535.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    R[d]<31:16> = imm16;
    // R[d]<15:0> unchanged
```

Exceptions

None.

A6.7.79 MRC, MRC2

Move to ARM Register from Coprocessor causes a coprocessor to transfer a value to an ARM register or to the condition flags.

Encoding T1 ARMv7-M

```
MRC<c> <coproc>,<opc1>,<Rt>,<CRn>,<CRm>{,<opc2>}
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	1	0	opc1			1	CRn				Rt				coproc				opc2			1	CRm			

```
t = UInt(Rt);  cp = UInt(coproc);
if t == 13 then UNPREDICTABLE;
```

Encoding T2 ARMv7-M

```
MRC2<c> <coproc>,<opc1>,<Rt>,<CRn>,<CRm>{,<opc2>}
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	0	opc1			1	CRn				Rt				coproc				opc2			1	CRm			

```
t = UInt(Rt);  cp = UInt(coproc);
if t == 13 then UNPREDICTABLE;
```

If no coprocessor can execute the instruction, a UsageFault exception is generated.

Assembler syntax

```
MRC{2}<c><q>  <coproc>, #<opc1>, <Rt>, <CRn>, <CRm>{, #<opc2>}
```

where:

`2`	If specified, selects the C == 1 form of the encoding. If omitted, selects the C == 0 form.
`<c><q>`	See *Standard assembler syntax fields* on page A6-7.
`<coproc>`	Specifies the name of the coprocessor. The standard generic coprocessor names are p0, p1, ..., p15.
`<opc1>`	Is a coprocessor-specific opcode in the range 0 to 7.
`<Rt>`	Is the destination ARM register. This register is allowed to be R0-R14 or APSR_nzcv. The last form writes bits<31:28> of the transferred value to the N, Z, C and V condition flags and is specified by setting the Rt field of the encoding to 0b1111. In pre-UAL assembler syntax, PC was written instead of APSR_nzcv to select this form.
`<CRn>`	Is the coprocessor register that contains the first operand.
`<CRm>`	Is an additional source or destination coprocessor register.
`<opc2>`	Is a coprocessor-specific opcode in the range 0 to 7. If it is omitted, `<opc2>` is assumed to be 0.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    if !Coproc_Accepted(cp, ThisInstr()) then
        GenerateCoprocessorException();
    else
        value = Coproc_GetOneWord(cp, ThisInstr());
        if t != 15 then
            R[t] = value;
        else
            APSR.N = value<31>;
            APSR.Z = value<30>;
            APSR.C = value<29>;
            APSR.V = value<28>;
            // value<27:0> are not used.
```

Exceptions

UsageFault.

A6.7.80 MRRC, MRRC2

Move to two ARM Registers from Coprocessor causes a coprocessor to transfer values to two ARM registers.

If no coprocessor can execute the instruction, a UsageFault exception is generated.

Encoding T1 ARMv7-M

`MRRC<c> <coproc>,<opc>,<Rt>,<Rt2>,<CRm>`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	0	0	0	1	0	1	Rt2				Rt				coproc				opc1				CRm			

```
t = UInt(Rt);  t2 = UInt(Rt2);  cp = UInt(coproc);
if t == 15 || t2 == 15 || t == t2 then UNPREDICTABLE;
if t == 13 || t2 == 13 then UNPREDICTABLE;
```

Encoding T2 ARMv7-M

`MRRC2<c> <coproc>,<opc>,<Rt>,<Rt2>,<CRm>`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	0	0	0	1	0	1	Rt2				Rt				coproc				opc1				CRm			

```
t = UInt(Rt);  t2 = UInt(Rt2);  cp = UInt(coproc);
if t == 15 || t2 == 15 || t == t2 then UNPREDICTABLE;
if t == 13 || t2 == 13 then UNPREDICTABLE;
```

Assembler syntax

```
MRRC{2}<c><q>  <coproc>, #<opc1>, <Rt>, <Rt2>, <CRm>
```

where:

`2` If specified, selects the C == 1 form of the encoding. If omitted, selects the C == 0 form.

`<c><q>` See *Standard assembler syntax fields* on page A6-7.

`<coproc>` Specifies the name of the coprocessor. The standard generic coprocessor names are p0, p1, ..., p15.

`<opc1>` Is a coprocessor-specific opcode in the range 0 to 15.

`<Rt>` Is the first destination ARM register.

`<Rt2>` Is the second destination ARM register.

`<CRm>` Is the coprocessor register that supplies the data to be transferred.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    if !Coproc_Accepted(cp, ThisInstr()) then
        GenerateCoprocessorException();
    else
        (R[t], R[t2]) = Coproc_GetTwoWords(cp, ThisInstr());
```

Exceptions

UsageFault.

A6.7.81 MRS

Move to Register from Special register moves the value from the selected special-purpose register into a general-purpose ARM register.

Encoding T1 ARMv6-M, ARMv7-M Enhanced functionality in ARMv7-M.

`MRS<c> <Rd>,<spec_reg>`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	1	1	(0)	(1)	(1)	(1)	(1)	1	0	(0)	0	Rd				SYSm							

Note

`MRS` is a system level instruction except when accessing the APSR (SYSm = 0) or CONTROL register (SYSm = 0x14). For the complete instruction definition see *MRS* on page B4-4.

A6.7.82 MSR (register)

Move to Special Register from ARM Register moves the value of a general-purpose ARM register to the specified special-purpose register.

Encoding T1 ARMv6-M, ARMv7-M Enhanced functionality in ARMv7-M.

`MSR<c> <spec_reg>,<Rn>`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	0	0	(0)	Rn				1	0	(0)	0	(1)	(0)	(0)	(0)	SYSm							

Note

`MSR`(register) is a system level instruction except when accessing the APSR (SYSm = 0). For the complete instruction definition see *MSR (register)* on page B4-8.

A6.7.83 MUL

Multiply multiplies two register values. The least significant 32 bits of the result are written to the destination register. These 32 bits do not depend on whether signed or unsigned calculations are performed.

It can optionally update the condition flags based on the result. This option is limited to only a few forms of the instruction in the Thumb instruction set, and use of it will adversely affect performance on many processor implementations.

Encoding T1 All versions of the Thumb ISA.

`MULS <Rdm>,<Rn>,<Rdm>` Outside IT block.

`MUL<c> <Rdm>,<Rn>,<Rdm>` Inside IT block.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	0	1	1	0	1	Rn			Rdm		

```
d = UInt(Rdm);  n = UInt(Rn);  m = UInt(Rdm);  setflags = !InITBlock();
```

Encoding T2 ARMv7-M

`MUL<c> <Rd>,<Rn>,<Rm>`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	1	1	0	0	0	0	Rn				1	1	1	1	Rd				0	0	0	0	Rm			

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  setflags = FALSE;
if d IN {13,15} || n IN {13,15} || m IN {13,15} then UNPREDICTABLE;
```

Assembler syntax

```
MUL{S}<c><q>  {<Rd>,} <Rn>, <Rm>
```

where:

S If present, specifies that the instruction updates the flags. Otherwise, the instruction does not update the flags.

<c><q> See *Standard assembler syntax fields* on page A6-7.

<Rd> Specifies the destination register. If <Rd> is omitted, this register is the same as <Rn>.

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

<Rm> Specifies the register that contains the second operand.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    operand1 = SInt(R[n]);  // operand1 = UInt(R[n]) produces the same final results
    operand2 = SInt(R[m]);  // operand2 = UInt(R[m]) produces the same final results
    result = operand1 * operand2;
    R[d] = result<31:0>;
    if setflags then
        APSR.N = result<31>;
        APSR.Z = IsZeroBit(result);
        // APSR.C unchanged
        // APSR.V unchanged
```

Exceptions

None.

A6.7.84 MVN (immediate)

Bitwise NOT (immediate) writes the bitwise inverse of an immediate value to the destination register. It can optionally update the condition flags based on the value.

Encoding T1 ARMv7-M

```
MVN{S}<c> <Rd>,#<const>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	i	0	0	0	1	1	S	1	1	1	1	0	imm3			Rd				imm8							

```
d = UInt(Rd);  setflags = (S == '1');
(imm32, carry) = ThumbExpandImm_C(i:imm3:imm8, APSR.C);
if d IN {13,15} then UNPREDICTABLE;
```

Assembler syntax

```
MVN{S}<c><q>  <Rd>, #<const>
```

where:

S	If present, specifies that the instruction updates the flags. Otherwise, the instruction does not update the flags.
<c><q>	See *Standard assembler syntax fields* on page A6-7.
<Rd>	Specifies the destination register.
<const>	Specifies the immediate value to be added to the value obtained from <Rn>. See *Modified immediate constants in Thumb instructions* on page A5-15 for the range of allowed values.

The pre-UAL syntax MVN<c>S is equivalent to MVNS<c>.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    result = NOT(imm32);
    R[d] = result;
    if setflags then
        APSR.N = result<31>;
        APSR.Z = IsZeroBit(result);
        APSR.C = carry;
        // APSR.V unchanged
```

Exceptions

None.

A6.7.85 MVN (register)

Bitwise NOT (register) writes the bitwise inverse of a register value to the destination register. It can optionally update the condition flags based on the result.

Encoding T1

All versions of the Thumb ISA.

`MVNS <Rd>,<Rm>`	Outside IT block.
`MVN<c> <Rd>,<Rm>`	Inside IT block.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	0	1	1	1	1	Rm			Rd		

```
d = UInt(Rd);  m = UInt(Rm);  setflags = !InITBlock();
(shift_t, shift_n) = (SRType_LSL, 0);
```

Encoding T2

ARMv7-M

`MVN{S}<c>.W <Rd>,<Rm>{,shift>}`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	0	1	0	0	1	1	S	1	1	1	1	(0)	imm3			Rd				imm2		type		Rm			

```
d = UInt(Rd);  m = UInt(Rm);  setflags = (S == '1');
(shift_t, shift_n) = DecodeImmShift(type, imm3:imm2);
if d IN {13,15} || m IN {13,15} then UNPREDICTABLE;
```

Assembler syntax

```
MVN{S}<c><q>  <Rd>, <Rm> {, <shift>}
```

where:

`S` If present, specifies that the instruction updates the flags. Otherwise, the instruction does not update the flags.

`<c><q>` See *Standard assembler syntax fields* on page A6-7.

`<Rd>` Specifies the destination register.

`<Rm>` Specifies the register that is optionally shifted and used as the source register.

`<shift>` Specifies the shift to apply to the value read from `<Rm>`. If `<shift>` is omitted, no shift is applied and both encodings are permitted. If `<shift>` is specified, only encoding T2 is permitted. The possible shifts and how they are encoded are described in *Shifts applied to a register* on page A6-12.

The pre-UAL syntax `MVN<c>S` is equivalent to `MVNS<c>`.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    (shifted, carry) = Shift_C(R[m], shift_t, shift_n, APSR.C);
    result = NOT(shifted);
    R[d] = result;
    if setflags then
        APSR.N = result<31>;
        APSR.Z = IsZeroBit(result);
        APSR.C = carry;
        // APSR.V unchanged
```

Exceptions

None.

A6.7.86 NEG

Negate is a pre-UAL synonym for `RSB` (immediate) with an immediate value of 0. See *RSB (immediate)* on page A6-200 for details.

Assembler syntax

```
NEG<c><q>  {<Rd>,} <Rm>
```

This is equivalent to:

```
RSBS<c><q>  {<Rd>,} <Rm>, #0
```

Exceptions

None.

A6.7.87 NOP

No Operation does nothing.

This is a NOP-compatible hint (the architected NOP), see *NOP-compatible hints* on page A6-16.

Encoding T1 ARMv7-M

```
NOP<c>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	1	1	1	0	0	0	0	0	0	0	0

```
// No additional decoding required
```

Encoding T2 ARMv7-M

```
NOP<c>.W
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	0	1	0	(1)	(1)	(1)	(1)	1	0	(0)	0	(0)	0	0	0	0	0	0	0	0	0	0	0

```
// No additional decoding required
```

Assembler syntax

```
NOP<c><q>
```

where:

`<c><q>` See *Standard assembler syntax fields* on page A6-7.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    // Do nothing
```

Exceptions

None.

A6.7.88 ORN (immediate)

Logical OR NOT (immediate) performs a bitwise (inclusive) OR of a register value and the complement of an immediate value, and writes the result to the destination register. It can optionally update the condition flags based on the result.

Encoding T1 ARMv7-M

`ORN{S}<c> <Rd>,<Rn>,#<const>`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	i	0	0	0	1	1	S	Rn				0	imm3			Rd				imm8							

```
if Rn == '1111' then SEE MVN (immediate);
d = UInt(Rd);  n = UInt(Rn);  setflags = (S == '1');
(imm32, carry) = ThumbExpandImm_C(i:imm3:imm8, APSR.C);
if d IN {13,15} || n == 13 then UNPREDICTABLE;
```

Assembler syntax

```
ORN{S}<c><q>  {<Rd>,} <Rn>, #<const>
```

where:

`S` If present, specifies that the instruction updates the flags. Otherwise, the instruction does not update the flags.

`<c><q>` See *Standard assembler syntax fields* on page A6-7.

`<Rd>` Specifies the destination register. If `<Rd>` is omitted, this register is the same as `<Rn>`.

`<Rn>` Specifies the register that contains the operand.

`<const>` Specifies the immediate value to be added to the value obtained from `<Rn>`. See *Modified immediate constants in Thumb instructions* on page A5-15 for the range of allowed values.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    result = R[n] OR NOT(imm32);
    R[d] = result;
    if setflags then
        APSR.N = result<31>;
        APSR.Z = IsZeroBit(result);
        APSR.C = carry;
        // APSR.V unchanged
```

Exceptions

None.

A6.7.89 ORN (register)

Logical OR NOT (register) performs a bitwise (inclusive) OR of a register value and the complement of an optionally-shifted register value, and writes the result to the destination register. It can optionally update the condition flags based on the result.

Encoding T1 ARMv7-M

```
ORN{S}<c> <Rd>,<Rn>,<Rm>{,<shift>}
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	0	1	0	0	1	1	S	Rn				(0)	imm3			Rd				imm2		type		Rm			

```
if Rn == '1111' then SEE MVN (register);
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  setflags = (S == '1');
(shift_t, shift_n) = DecodeImmShift(type, imm3:imm2);
if d IN {13,15} || n == 13 || m IN {13,15} then UNPREDICTABLE;
```

Assembler syntax

```
ORN{S}<c><q>  {<Rd>,} <Rn>, <Rm> {,<shift>}
```

where:

`S` If present, specifies that the instruction updates the flags. Otherwise, the instruction does not update the flags.

`<c><q>` See *Standard assembler syntax fields* on page A6-7.

`<Rd>` Specifies the destination register. If `<Rd>` is omitted, this register is the same as `<Rn>`.

`<Rn>` Specifies the register that contains the first operand.

`<Rm>` Specifies the register that is optionally shifted and used as the second operand.

`<shift>` Specifies the shift to apply to the value read from `<Rm>`. If `<shift>` is omitted, no shift is applied. The possible shifts and how they are encoded are described in *Shifts applied to a register* on page A6-12.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    (shifted, carry) = Shift_C(R[m], shift_t, shift_n, APSR.C);
    result = R[n] OR NOT(shifted);
    R[d] = result;
    if setflags then
        APSR.N = result<31>;
        APSR.Z = IsZeroBit(result);
        APSR.C = carry;
        // APSR.V unchanged
```

Exceptions

None.

A6.7.90 ORR (immediate)

Logical OR (immediate) performs a bitwise (inclusive) OR of a register value and an immediate value, and writes the result to the destination register. It can optionally update the condition flags based on the result.

Encoding T1 ARMv7-M

```
ORR{S}<c> <Rd>,<Rn>,#<const>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	i	0	0	0	1	0	S	Rn				0	imm3			Rd				imm8							

```
if Rn == '1111' then SEE MOV (immediate);
d = UInt(Rd);  n = UInt(Rn);  setflags = (S == '1');
(imm32, carry) = ThumbExpandImm_C(i:imm3:imm8, APSR.C);
if d IN {13,15} || n == 13 then UNPREDICTABLE;
```

Assembler syntax

```
ORR{S}<c><q>  {<Rd>,} <Rn>, #<const>
```

where:

S If present, specifies that the instruction updates the flags. Otherwise, the instruction does not update the flags.

<c><q> See *Standard assembler syntax fields* on page A6-7.

<Rd> Specifies the destination register. If <Rd> is omitted, this register is the same as <Rn>.

<Rn> Specifies the register that contains the operand.

<const> Specifies the immediate value to be added to the value obtained from <Rn>. See *Modified immediate constants in Thumb instructions* on page A5-15 for the range of allowed values.

The pre-UAL syntax ORR<c>S is equivalent to ORRS<c>.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    result = R[n] OR imm32;
    R[d] = result;
    if setflags then
        APSR.N = result<31>;
        APSR.Z = IsZeroBit(result);
        APSR.C = carry;
        // APSR.V unchanged
```

Exceptions

None.

A6.7.91 ORR (register)

Logical OR (register) performs a bitwise (inclusive) OR of a register value and an optionally-shifted register value, and writes the result to the destination register. It can optionally update the condition flags based on the result.

Encoding T1

All versions of the Thumb ISA.

`ORRS <Rdn>,<Rm>`	Outside IT block.
`ORR<c> <Rdn>,<Rm>`	Inside IT block.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	0	1	1	0	0	Rm			Rdn		

```
d = UInt(Rdn);  n = UInt(Rdn);  m = UInt(Rm);  setflags = !InITBlock();
(shift_t, shift_n) = (SRType_LSL, 0);
```

Encoding T2

ARMv7-M

`ORR{S}<c>.W <Rd>,<Rn>,<Rm>{,<shift>}`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	0	1	0	0	1	0	S	Rn				(0)	imm3			Rd				imm2		type		Rm			

```
if Rn == '1111' then SEE MOV (register);
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  setflags = (S == '1');
(shift_t, shift_n) = DecodeImmShift(type, imm3:imm2);
if d IN {13,15} || n == 13 || m IN {13,15} then UNPREDICTABLE;
```

Assembler syntax

```
ORR{S}<c><q>  {<Rd>,} <Rn>, <Rm> {,<shift>}
```

where:

`S` If present, specifies that the instruction updates the flags. Otherwise, the instruction does not update the flags.

`<c><q>` See *Standard assembler syntax fields* on page A6-7.

`<Rd>` Specifies the destination register. If `<Rd>` is omitted, this register is the same as `<Rn>`.

`<Rn>` Specifies the register that contains the first operand.

`<Rm>` Specifies the register that is optionally shifted and used as the second operand.

`<shift>` Specifies the shift to apply to the value read from `<Rm>`. If `<shift>` is omitted, no shift is applied and both encodings are permitted. If `<shift>` is specified, only encoding T2 is permitted. The possible shifts and how they are encoded are described in *Shifts applied to a register* on page A6-12.

A special case is that if `ORR<c> <Rd>,<Rn>,<Rd>` is written with `<Rd>` and `<Rn>` both in the range R0-R7, it will be assembled using encoding T2 as though `ORR<c> <Rd>,<Rn>` had been written. To prevent this happening, use the .W qualifier.

The pre-UAL syntax `ORR<c>S` is equivalent to `ORRS<c>`.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    (shifted, carry) = Shift_C(R[m], shift_t, shift_n, APSR.C);
    result = R[n] OR shifted;
    R[d] = result;
    if setflags then
        APSR.N = result<31>;
        APSR.Z = IsZeroBit(result);
        APSR.C = carry;
        // APSR.V unchanged
```

Exceptions

None.

A6.7.92 PLD~~, PLDW~~ (immediate)

Preload Data signals the memory system that data memory accesses from a specified address are likely in the near future. The memory system can respond by taking actions that are expected to speed up the memory accesses when they do occur, such as pre-loading the cache line containing the specified address into the data cache. See *Preloading caches* on page A3-39 and *Memory hints* on page A6-16 for additional information.

~~Where both the PLD and PLDW instructions are implemented, the PLD instruction signals that the likely memory access is a read, and the PLDW instruction signals that it is a write.~~

Encoding T1 ARMv7-M

PLD~~{W}~~<c> [<Rn>,#<imm12>]

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	0	1	0	~~W~~	1	Rn				1	1	1	1	imm12											

```
if Rn == '1111' then SEE PLD (literal);
n = UInt(Rn);  imm32 = ZeroExtend(imm12, 32);  add = TRUE;~~ is_pldw = (W == '1');~~
```

Encoding T2 ARMv7-M

PLD~~{W}~~<c> [<Rn>,#-<imm8>]

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	0	0	0	~~W~~	1	Rn				1	1	1	1	1	1	0	0	imm8							

```
if Rn == '1111' then SEE PLD (literal);
n = UInt(Rn);  imm32 = ZeroExtend(imm8, 32);  add = FALSE;~~ is_pldw = (W == '1');~~
```

Assembler syntax

```
PLD~~{W}~~<c><q>  [<Rn> {, #+/-<imm>}]
```

where:

~~`W`~~ ~~If specified, selects PLDW, encoded as W = 1. If omitted, selects PLD, encoded as W = 0.~~

`<c><q>` See *Standard assembler syntax fields* on page A6-7.

`<Rn>` The base register. The SP can be used. For PC use in the `PLD` instruction, see *PLD (literal)* on page A6-178.

`+/-` Is + or omitted to indicate that the immediate offset is added to the base register value (`add == TRUE`), or – to indicate that the offset is to be subtracted (`add == FALSE`). Different instructions are generated for `#0` and `#-0`.

`<imm>` The immediate offset used to form the address. This offset can be omitted, meaning an offset of 0. Values are:

Encoding T1 any value in the range 0-4095

Encoding T2 any value in the range 0-255.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    address = if add then (R[n] + imm32) else (R[n] - imm32);
    ~~if is_pldw then~~
        ~~Hint_PreloadDataForWrite(address);~~
    ~~else~~
        Hint_PreloadData(address);
```

Exceptions

None.

A6.7.93 PLD (literal)

Preload Data signals the memory system that data memory accesses from a specified address are likely in the near future. The memory system can respond by taking actions that are expected to speed up the memory accesses when they do occur, such as pre-loading the cache line containing the specified address into the data cache. See *Preloading caches* on page A3-39 and *Memory hints* on page A6-16 for additional information.

Encoding T1 ARMv7-M

```
PLD<c> <label>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	0	U	0	0	1	1	1	1	1	1	1	1	1	imm12											

```
imm32 = ZeroExtend(imm12, 32);  add = (U == '1');
```

Assembler syntax

`PLD<c><q> <label>`	Normal form
`PLD<c><q> [PC, #+/-<imm>]`	Alternative form

where:

`<c><q>` See *Standard assembler syntax fields* on page A6-7.

`<label>` The label of the literal item that is likely to be accessed in the near future. The assembler calculates the required value of the offset from the PC value of this instruction to the label. The offset must be in the range -4095 to 4095.

If the offset is zero or positive, `imm32` is equal to the offset and `add == TRUE`

If the offset is negative, `imm32` is equal to minus the offset and `add == FALSE`

`+/-` Is + or omitted to indicate that the immediate offset is added to the base register value (`add == TRUE`), or – to indicate that the offset is to be subtracted (`add == FALSE`). Different instructions are generated for `#0` and `#-0`.

`<imm>` The immediate offset used to form the address. Values are in the range 0-4095.

The alternative syntax permits the addition or subtraction of the offset and the immediate offset to be specified separately, including permitting a subtraction of 0 that cannot be specified using the normal syntax. For more information, see *Use of labels in UAL instruction syntax* on page A4-5.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    address = if add then (Align(PC,4) + imm32) else (Align(PC,4) - imm32);
    Hint_PreloadData(address);
```

Exceptions

None.

A6.7.94 PLD (register)

Preload Data is a memory hint instruction that can signal the memory system that data memory accesses from a specified address are likely in the near future. The memory system can respond by taking actions that are expected to speed up the memory accesses when they do occur, such as pre-loading the cache line containing the specified address into the data cache. See *Preloading caches* on page A3-39 and *Memory hints* on page A6-16 for additional information.

Encoding T1 ARMv7-M

`PLD<c> [<Rn>,<Rm>{,LSL #<imm2>}]`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	0	0	0	0	1	Rn				1	1	1	1	0	0	0	0	0	0	shift		Rm			

```
if Rn == '1111' then SEE PLD (literal);
n = UInt(Rn);  m = UInt(Rm);  add = TRUE;~~ is_pldw = (W == '1');~~
(shift_t, shift_n) = (SRType_LSL, UInt(imm2));
if m IN {13,15} then UNPREDICTABLE;
```

Assembler syntax

```
PLD<c><q>  [<Rn>, <Rm> {, LSL #<shift>}]
```

where:

`<c><q>`	See *Standard assembler syntax fields* on page A6-7.
`<Rn>`	Is the base register. This register is allowed to be the SP.
`<Rm>`	Is the optionally shifted offset register.
`<shift>`	Specifies the shift to apply to the value read from `<Rm>`, in the range 0-3. If this option is omitted, a shift by 0 is assumed.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    offset = Shift(R[m], shift_t, shift_n, APSR.C);
    address = if add then (R[n] + offset) else (R[n] - offset);
    Hint_PreloadData(address);
```

Exceptions

None.

A6.7.95 PLI (immediate, literal)

Preload Instruction is a memory hint instruction that can signal the memory system that instruction memory accesses from a specified address are likely in the near future. The memory system can respond by taking actions that are expected to speed up the memory accesses when they do occur, such as pre-loading the cache line containing the specified address into the instruction cache. See *Preloading caches* on page A3-39 and *Memory hints* on page A6-16 for additional information.

Encoding T1 ARMv7

```
PLI<c> [<Rn>,#<imm12>]
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	1	1	0	0	1	Rn				1	1	1	1	imm12											

```
if Rn == '1111' then SEE encoding T3;
n = UInt(Rn);  imm32 = ZeroExtend(imm12, 32);  add = TRUE;
```

Encoding T2 ARMv7

```
PLI<c> [<Rn>,#-<imm8>]
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	1	0	0	0	1	Rn				1	1	1	1	1	1	0	0	imm8							

```
if Rn == '1111' then SEE encoding T3;
n = UInt(Rn);  imm32 = ZeroExtend(imm8, 32);  add = FALSE;
```

Encoding T3 ARMv7

```
PLI<c> <label>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	1	U	0	0	1	1	1	1	1	1	1	1	1	imm12											

```
n = 15;  imm32 = ZeroExtend(imm12, 32);  add = (U == '1');
```

Assembler syntax

```
PLI<c><q>  [<Rn>, #+/-<imm>]
PLI<c><q>  [PC, #+/-<imm>]
```

where:

`<c><q>` See *Standard assembler syntax fields* on page A6-7.

`<Rn>` Is the base register. This register is allowed to be the SP.

`+/-` Is + or omitted to indicate that the immediate offset is added to the base register value (`add == TRUE`), or – to indicate that the offset is to be subtracted (`add == FALSE`). Different instructions are generated for `#0` and `#-0`.

`<imm>` Specifies the offset from the base register. It must be in the range:

- –4095 to 4095 if the base register is the PC
- –255 to 4095 otherwise.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    base = if n == 15 then Align(PC,4) else R[n];
    address = if add then (base + imm32) else (base - imm32);
    Hint_PreloadInstr(address);
```

Exceptions

None.

A6.7.96 PLI (register)

Preload Instruction is a memory hint instruction that can signal the memory system that instruction memory accesses from a specified address are likely in the near future. The memory system can respond by taking actions that are expected to speed up the memory accesses when they do occur, such as pre-loading the cache line containing the specified address into the instruction cache. See *Preloading caches* on page A3-39 and *Memory hints* on page A6-16 for additional information.

Encoding T1 ARMv7

```
PLI<c> [<Rn>,<Rm>{,LSL #<imm2>}]
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	1	0	0	0	1	Rn				1	1	1	1	0	0	0	0	0	0	shift		Rm			

```
if Rn == '1111' then SEE PLI (immediate, literal);
n = UInt(Rn);  m = UInt(Rm);  add = TRUE;
(shift_t, shift_n) = (SRType_LSL, UInt(imm2));
if m IN {13,15} then UNPREDICTABLE;
```

Assembler syntax

```
PLI<c><q>  [<Rn>, <Rm> {, LSL #<shift>}]
```

where:

`<c><q>` See *Standard assembler syntax fields* on page A6-7.

`<Rn>` Is the base register. This register is allowed to be the SP.

`<Rm>` Is the optionally shifted offset register.

`<shift>` Specifies the shift to apply to the value read from `<Rm>`, in the range 0-3. If this option is omitted, a shift by 0 is assumed.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    offset = Shift(R[m], shift_t, shift_n, APSR.C);
    address = if add then (R[n] + offset) else (R[n] - offset);
    Hint_PreloadInstr(address);
```

Exceptions

None.

A6.7.97 POP

Pop Multiple Registers loads a subset (or possibly all) of the general-purpose registers R0-R12 and the PC or the LR from the stack.

If the registers loaded include the PC, the word loaded for the PC is treated as ~~an address or an exception return value and a branch occurs.~~ Bit<0> complies with the ARM architecture interworking rules for branches to Thumb state execution and must be 1. If bit<0> is 0, a UsageFault exception occurs.

Encoding T1 All versions of the Thumb ISA.

```
POP<c> <registers>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	1	0	P	register_list							

```
registers = P:'0000000':register_list;  if BitCount(registers) < 1 then UNPREDICTABLE;
```

Encoding T2 ARMv7-M

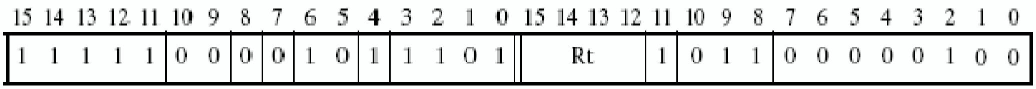
```
POP<c>.W <registers>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	0	0	0	1	0	1	1	1	1	0	1	P	M	(0)	register_list												

```
registers = P:M:'0':register_list;
if BitCount(registers) < 2 || (P == '1' && M == '1') then UNPREDICTABLE;
if registers<15> == '1' && InITBlock() && !LastInITBlock() then UNPREDICTABLE;
```

Encoding T3 ARMv7-M

POP<c>.W <registers> contains one register, <Rt>



15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	0	0	1	0	1	1	1	0	1	Rt				1	0	1	1	0	0	0	0	0	1	0	0

```
t = UInt(Rt);  registers = Zeros(16);  registers<t> = '1';
if t == 13 || (t == 15 && InITBlock() && !LastInITBlock()) then UNPREDICTABLE;
```

Assembler syntax

`POP<c><q>  <registers>`	Standard syntax
`LDMIA<c><q>  SP!, <registers>`	Equivalent `LDM` syntax

where:

`<c><q>` See *Standard assembler syntax fields* on page A6-7.

`<registers>`

Is a list of one or more registers, separated by commas and surrounded by { and }. It specifies the set of registers to be loaded. The registers are loaded in sequence, the lowest-numbered register from the lowest memory address, through to the highest-numbered register from the highest memory address. If the PC is specified in the register list, the instruction causes a branch to the address (data) loaded into the PC.

~~Encoding T2 does not support a list containing only one register. If a `POP` instruction with just one register `<Rt>` in the list is assembled to Thumb and encoding T1 is not available, it is assembled to the equivalent `LDR<c><q> <Rt>,[SP],#-4` instruction.~~

The SP cannot be in the list.

If the PC is in the list, the LR must not be in the list.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    address = SP;

    for i = 0 to 14
        if registers<i> == '1' then
            R[i} = MemA[address,4];  address = address + 4;
    if registers<15> == '1' then
        LoadWritePC(MemA[address,4]);

    SP = SP + 4*BitCount(registers);
```

Exceptions

UsageFault, MemManage, BusFault.

A6.7.98 PUSH

Push Multiple Registers stores a subset (or possibly all) of the general-purpose registers R0-R12 and the LR to the stack.

Encoding T1 All versions of the Thumb ISA.

```
PUSH<c> <registers>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	0	1	0	M	register_list							

```
registers = '0':M:'000000':register_list;
if BitCount(registers) < 1 then UNPREDICTABLE;
```

Encoding T2 ARMv7-M

```
PUSH<c>.W <registers>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	0	0	1	0	0	1	0	1	1	0	1	(0)	M	(0)	register_list												

```
registers = '0':M:'0':register_list;
if BitCount(registers) < 2 then UNPREDICTABLE;
```

Encoding T3 ARMv7-M

```
PUSH<c>.W <registers>
```
<registers> contains one register, <Rt>

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	0	0	1	0	0	1	1	0	1	Rt				1	1	0	1	0	0	0	0	0	1	0	0

```
t = UInt(Rt);  registers = Zeros(16);  registers<t> = '1';
if t == 13 || t == 15  then UNPREDICTABLE;
```

Assembler syntax

`PUSH<c><q> <registers>`	Standard syntax
`STMDB<c><q> SP!, <registers>`	Equivalent `STM` syntax

where:

`<c><q>` See *Standard assembler syntax fields* on page A6-7.

`<registers>`

Is a list of one or more registers, separated by commas and surrounded by { and }. It specifies the set of registers to be stored. The registers are stored in sequence, the lowest-numbered register to the lowest memory address, through to the highest-numbered register to the highest memory address.

~~Encoding T2 does not support a list containing only one register. If a `PUSH` instruction with just one register `<Rt>` in the list is assembled to Thumb and encoding T1 is not available, it is assembled to the equivalent `STR<c><q> <Rt>,[SP,#-4]!` instruction.~~

The SP and PC cannot be in the list.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    address = SP - 4*BitCount(registers);

    for i = 0 to 14
        if registers<i> == '1' then
            MemA[address,4] = R[i];
            address = address + 4;

    SP = SP - 4*BitCount(registers);
```

Exceptions

UsageFault, MemManage, BusFault.

A6.7.99 RBIT

Reverse Bits reverses the bit order in a 32-bit register.

Encoding T1 ARMv7-M

`RBIT<c> <Rd>,<Rm>`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	1	0	1	0	0	1	Rm				1	1	1	1	Rd				1	0	1	0	Rm			

```
if !Consistent(Rm) then UNPREDICTABLE;
d = UInt(Rd);  m = UInt(Rm);
if d IN {13,15} || m IN {13,15} then UNPREDICTABLE;
```

Assembler syntax

`RBIT<c><q>  <Rd>, <Rm>`

where:

`<c><q>` See *Standard assembler syntax fields* on page A6-7.

`<Rd>` Specifies the destination register.

`<Rm>` Specifies the register that contains the operand. Its number must be encoded twice in encoding T1, in both the Rm and Rm2 fields.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    bits(32) result;
    for i = 0 to 31 do
        result<31-i> = R[m]<i>;
    R[d] = result;
```

Exceptions

None.

A6.7.100 REV

Byte-Reverse Word reverses the byte order in a 32-bit register.

Encoding T1 ARMv6-M, ARMv7-M

`REV<c> <Rd>,<Rm>`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	0	1	0	0	0	Rm			Rd		

```
d = UInt(Rd);  m = UInt(Rm);
```

Encoding T2 ARMv7-M

`REV<c>.W <Rd>,<Rm>`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	1	0	1	0	0	1	Rm				1	1	1	1	Rd				1	0	0	0	Rm			

```
if !Consistent(Rm) then UNPREDICTABLE;
d = UInt(Rd);  m = UInt(Rm);
if d IN {13,15} || m IN {13,15} then UNPREDICTABLE;
```

Assembler syntax

`REV<c><q>  <Rd>, <Rm>`

where:

`<c><q>` See *Standard assembler syntax fields* on page A6-7.

`<Rd>` Specifies the destination register.

`<Rm>` Specifies the register that contains the operand. Its number must be encoded twice in encoding T2, in both the Rm and Rm2 fields.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    bits(32) result;
    result<31:24> = R[m]<7:0>;
    result<23:16> = R[m]<15:8>;
    result<15:8>  = R[m]<23:16>;
    result<7:0>   = R[m]<31:24>;
    R[d] = result;
```

Exceptions

None.

A6.7.101 REV16

Byte-Reverse Packed Halfword reverses the byte order in each 16-bit halfword of a 32-bit register.

Encoding T1 ARMv6-M, ARMv7-M

```
REV16<c> <Rd>,<Rm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	0	1	0	0	1	Rm			Rd		

```
d = UInt(Rd);  m = UInt(Rm);
```

Encoding T2 ARMv7-M

```
REV16<c>.W <Rd>,<Rm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	1	0	1	0	0	1	Rm				1	1	1	1	Rd				1	0	0	1	Rm			

```
if !Consistent(Rm) then UNPREDICTABLE;
d = UInt(Rd);  m = UInt(Rm);
if d IN {13,15} || m IN {13,15} then UNPREDICTABLE;
```

Assembler syntax

```
REV16<c><q>  <Rd>, <Rm>
```

where:

`<c><q>` See *Standard assembler syntax fields* on page A6-7.

`<Rd>` Specifies the destination register.

`<Rm>` Specifies the register that contains the operand. Its number must be encoded twice in encoding T2, in both the Rm and Rm2 fields.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    bits(32) result;
    result<31:24> = R[m]<23:16>;
    result<23:16> = R[m]<31:24>;
    result<15:8>  = R[m]<7:0>;
    result<7:0>   = R[m]<15:8>;
    R[d] = result;
```

Exceptions

None.

A6.7.102 REVSH

Byte-Reverse Signed Halfword reverses the byte order in the lower 16-bit halfword of a 32-bit register, and sign extends the result to 32 bits.

Encoding T1 ARMv6-M, ARMv7-M

`REVSH<c> <Rd>,<Rm>`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	0	1	0	1	1	Rm			Rd		

```
d = UInt(Rd);  m = UInt(Rm);
```

Encoding T2 ARMv7-M

`REVSH<c>.W <Rd>,<Rm>`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	1	0	1	0	0	1	Rm				1	1	1	1	Rd				1	0	1	1	Rm			

```
if !Consistent(Rm) then UNPREDICTABLE;
d = UInt(Rd);  m = UInt(Rm);
if d IN {13,15} || m IN {13,15} then UNPREDICTABLE;
```

Assembler syntax

```
REVSH<c><q>  <Rd>, <Rm>
```

where:

`<c><q>` See *Standard assembler syntax fields* on page A6-7.

`<Rd>` Specifies the destination register.

`<Rm>` Specifies the register that contains the operand. Its number must be encoded twice in encoding T2, in both the Rm and Rm2 fields.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    bits(32) result;
    result<31:8>  = SignExtend(R[m]<7:0>, 24);
    result<7:0>   = R[m]<15:8>;
    R[d] = result;
```

Exceptions

None.

A6.7.103 ROR (immediate)

Rotate Right (immediate) provides the value of the contents of a register rotated by a constant value. The bits that are rotated off the right end are inserted into the vacated bit positions on the left. It can optionally update the condition flags based on the result.

Encoding T1 ARMv7-M

```
ROR{S}<c> <Rd>,<Rm>,#<imm5>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	0	1	0	0	1	0	S	1	1	1	1	(0)	imm3			Rd				imm2		1	1	Rm			

```
if (imm3:imm2) == '00000' then SEE RRX;
d = UInt(Rd);  m = UInt(Rm);  setflags = (S == '1');
(-, shift_n) = DecodeImmShift('11', imm3:imm2);
if d IN {13,15} || m IN {13,15} then UNPREDICTABLE;
```

Assembler syntax

```
ROR{S}<c><q>  <Rd>, <Rm>, #<imm5>
```

where:

`S` If present, specifies that the instruction updates the flags. Otherwise, the instruction does not update the flags.

`<c><q>` See *Standard assembler syntax fields* on page A6-7.

`<Rd>` Specifies the destination register.

`<Rm>` Specifies the register that contains the first operand.

`<imm5>` Specifies the shift amount, in the range 1 to 31. See *Shifts applied to a register* on page A6-12.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    (result, carry) = Shift_C(R[m], SRType_ROR, shift_n, APSR.C);
    R[d] = result;
    if setflags then
        APSR.N = result<31>;
        APSR.Z = IsZeroBit(result);
        APSR.C = carry;
        // APSR.V unchanged
```

Exceptions

None.

A6.7.104 ROR (register)

Rotate Right (register) provides the value of the contents of a register rotated by a variable number of bits. The bits that are rotated off the right end are inserted into the vacated bit positions on the left. The variable number of bits is read from the bottom byte of a register. It can optionally update the condition flags based on the result.

Encoding T1 All versions of the Thumb ISA.

`RORS <Rdn>,<Rm>` Outside IT block.

`ROR<c> <Rdn>,<Rm>` Inside IT block.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	0	0	1	1	1	Rm			Rdn		

```
d = UInt(Rdn);  n = UInt(Rdn);  m = UInt(Rm);  setflags = !InITBlock();
```

Encoding T2 ARMv7-M

`ROR{S}<c>.W <Rd>,<Rn>,<Rm>`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	1	0	0	1	1	S	Rn				1	1	1	1	Rd				0	0	0	0	Rm			

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  setflags = (S == '1');
if d IN {13,15} || n IN {13,15} || m IN {13,15} then UNPREDICTABLE;
```

Assembler syntax

```
ROR{S}<c><q>  <Rd>, <Rn>, <Rm>
```

where:

`S` If present, specifies that the instruction updates the flags. Otherwise, the instruction does not update the flags.

`<c><q>` See *Standard assembler syntax fields* on page A6-7.

`<Rd>` Specifies the destination register.

`<Rn>` Specifies the register that contains the first operand.

`<Rm>` Specifies the register whose bottom byte contains the amount to rotate by.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    shift_n = UInt(R[m]<7:0>);
    (result, carry) = Shift_C(R[n], SRType_ROR, shift_n, APSR.C);
    R[d] = result;
    if setflags then
        APSR.N = result<31>;
        APSR.Z = IsZeroBit(result);
        APSR.C = carry;
        // APSR.V unchanged
```

Exceptions

None.

A6.7.105 RRX

Rotate Right with Extend provides the value of the contents of a register shifted right by one place, with the carry flag shifted into bit<31>.

RRX can optionally update the condition flags based on the result. In that case, bit<0> is shifted into the carry flag.

Encoding T1 ARMv7-M

RRX{S}<c> <Rd>,<Rm>

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	0	1	0	0	1	0	S	1	1	1	1	(0)	0	0	0	Rd				0	0	1	1	Rm			

```
d = UInt(Rd);  m = UInt(Rm);  setflags = (S == '1');
if d IN {13,15} || m IN {13,15} then UNPREDICTABLE;
```

Assembler syntax

```
RRX{S}<c><q>  <Rd>, <Rm>
```

where:

S If present, specifies that the instruction updates the flags. Otherwise, the instruction does not update the flags.

<c><q> See *Standard assembler syntax fields* on page A6-7.

<Rd> Specifies the destination register.

<Rm> Specifies the register that contains the operand.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    (result, carry) = Shift_C(R[m], SRType_RRX, 1, APSR.C);
    R[d] = result;
    if setflags then
        APSR.N = result<31>;
        APSR.Z = IsZeroBit(result);
        APSR.C = carry;
        // APSR.V unchanged
```

Exceptions

None.

A6.7.106 RSB (immediate)

Reverse Subtract (immediate) subtracts a register value from an immediate value, and writes the result to the destination register. It can optionally update the condition flags based on the result.

Encoding T1 All versions of the Thumb ISA.

`RSBS <Rd>,<Rn>,#0` Outside IT block.

`RSB<c> <Rd>,<Rn>,#0` Inside IT block.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	0	1	0	0	1	Rn			Rd		

```
d = UInt(Rd);  n = UInt(Rn);  setflags = !InITBlock();  imm32 = Zeros(32); // immediate = #0
```

Encoding T2 ARMv7-M

`RSB{S}<c>.W <Rd>,<Rn>,#<const>`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	i	0	1	1	1	0	S	Rn				0	imm3			Rd				imm8							

```
d = UInt(Rd);  n = UInt(Rn);  setflags = (S == '1');  imm32 = ThumbExpandImm(i:imm3:imm8);
if d IN {13,15} || n IN {13,15} then UNPREDICTABLE;
```

Assembler syntax

```
RSB{S}<c><q>  {<Rd>,} <Rn>, #<const>
```

where:

`S`	If present, specifies that the instruction updates the flags. Otherwise, the instruction does not update the flags.
`<c><q>`	See *Standard assembler syntax fields* on page A6-7.
`<Rd>`	Specifies the destination register. If `<Rd>` is omitted, this register is the same as `<Rn>`.
`<Rn>`	Specifies the register that contains the first operand.
`<const>`	Specifies the immediate value to be added to the value obtained from `<Rn>`. The only allowed value for encoding T1 is 0. See *Modified immediate constants in Thumb instructions* on page A5-15 for the range of allowed values for encoding T2.

The pre-UAL syntax `RSB<c>S` is equivalent to `RSBS<c>`.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    (result, carry, overflow) = AddWithCarry(NOT(R[n]), imm32, ’1’);
    R[d] = result;
    if setflags then
        APSR.N = result<31>;
        APSR.Z = IsZeroBit(result);
        APSR.C = carry;
        APSR.V = overflow;
```

Exceptions

None.

A6.7.107 RSB (register)

Reverse Subtract (register) subtracts a register value from an optionally-shifted register value, and writes the result to the destination register. It can optionally update the condition flags based on the result.

Encoding T1 ARMv7-M

`RSB{S}<c> <Rd>,<Rn>,<Rm>{,<shift>}`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	0	1	1	1	1	0	S	Rn				(0)	imm3			Rd				imm2		type		Rm			

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  setflags = (S == '1');
(shift_t, shift_n) = DecodeImmShift(type, imm3:imm2);
if d IN {13,15} || n IN {13,15} || m IN {13,15} then UNPREDICTABLE;
```

Assembler syntax

```
RSB{S}<c><q>  {<Rd>,} <Rn>, <Rm> {,<shift>}
```

where:

`S` If present, specifies that the instruction updates the flags. Otherwise, the instruction does not update the flags.

`<c><q>` See *Standard assembler syntax fields* on page A6-7.

`<Rd>` Specifies the destination register. If `<Rd>` is omitted, this register is the same as `<Rn>`.

`<Rn>` Specifies the register that contains the first operand.

`<Rm>` Specifies the register that is optionally shifted and used as the second operand.

`<shift>` Specifies the shift to apply to the value read from `<Rm>`. If `<shift>` is omitted, no shift is applied. The possible shifts and how they are encoded are described in *Shifts applied to a register* on page A6-12.

The pre-UAL syntax `RSB<c>S` is equivalent to `RSBS<c>`.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    shifted = Shift(R[m], shift_t, shift_n, APSR.C);
    (result, carry, overflow) = AddWithCarry(NOT(R[n]), shifted, '1');
    R[d] = result;
    if setflags then
        APSR.N = result<31>;
        APSR.C = carry;
        APSR.V = overflow;
```

Exceptions

None.

A6.7.108 SBC (immediate)

Subtract with Carry (immediate) subtracts an immediate value and the value of NOT(Carry flag) from a register value, and writes the result to the destination register. It can optionally update the condition flags based on the result.

Encoding T1 ARMv7-M

```
SBC{S}<c> <Rd>,<Rn>,#<const>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	i	0	1	0	1	1	S	Rn				0	imm3			Rd				imm8							

```
d = UInt(Rd);  n = UInt(Rn);  setflags = (S == '1');  imm32 = ThumbExpandImm(i:imm3:imm8);
if d IN {13,15} || n IN {13,15} then UNPREDICTABLE;
```

Assembler syntax

```
SBC{S}<c><q>  {<Rd>,} <Rn>, #<const>
```

where:

S	If present, specifies that the instruction updates the flags. Otherwise, the instruction does not update the flags.

<c><q>	See *Standard assembler syntax fields* on page A6-7.

<Rd>	Specifies the destination register. If <Rd> is omitted, this register is the same as <Rn>.

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

<const>	Specifies the immediate value to be added to the value obtained from <Rn>. See *Modified immediate constants in Thumb instructions* on page A5-15 for the range of allowed values.

The pre-UAL syntax SBC<c>S is equivalent to SBCS<c>.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    (result, carry, overflow) = AddWithCarry(R[n], NOT(imm32), APSR.C);
    R[d] = result;
    if setflags then
        APSR.N = result<31>;
        APSR.Z = IsZeroBit(result);
        APSR.C = carry;
        APSR.V = overflow;
```

Exceptions

None.

A6.7.109 SBC (register)

Subtract with Carry (register) subtracts an optionally-shifted register value and the value of NOT(Carry flag) from a register value, and writes the result to the destination register. It can optionally update the condition flags based on the result.

Encoding T1

All versions of the Thumb ISA.

`SBCS <Rdn>,<Rm>`	Outside IT block.
`SBC<c> <Rdn>,<Rm>`	Inside IT block.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	0	0	1	1	0	Rm			Rdn		

```
d = UInt(Rdn);  n = UInt(Rdn);  m = UInt(Rm);  setflags = !InITBlock();
(shift_t, shift_n) = (SRType_LSL, 0);
```

Encoding T2

ARMv7-M

`SBC{S}<c>.W <Rd>,<Rn>,<Rm>{,<shift>}`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	0	1	1	0	1	1	S	Rn				(0)	imm3			Rd				imm2		type		Rm			

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  setflags = (S == '1');
(shift_t, shift_n) = DecodeImmShift(type, imm3:imm2);
if d IN {13,15} || n IN {13,15} || m IN {13,15} then UNPREDICTABLE;
```

Assembler syntax

```
SBC{S}<c><q>  {<Rd>,} <Rn>, <Rm> {,<shift>}
```

where:

`S` If present, specifies that the instruction updates the flags. Otherwise, the instruction does not update the flags.

`<c><q>` See *Standard assembler syntax fields* on page A6-7.

`<Rd>` Specifies the destination register. If `<Rd>` is omitted, this register is the same as `<Rn>`.

`<Rn>` Specifies the register that contains the first operand.

`<Rm>` Specifies the register that is optionally shifted and used as the second operand.

`<shift>` Specifies the shift to apply to the value read from `<Rm>`. If `<shift>` is omitted, no shift is applied and both encodings are permitted. If <shift> is specified, only encoding T2 is permitted. The possible shifts and how they are encoded are described in *Shifts applied to a register* on page A6-12.

The pre-UAL syntax `SBC<c>S` is equivalent to `SBCS<c>`.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    shifted = Shift(R[m], shift_t, shift_n, APSR.C);
    (result, carry, overflow) = AddWithCarry(R[n], NOT(shifted), APSR.C);
    R[d] = result;
    if setflags then
        APSR.N = result<31>;
        APSR.Z = IsZeroBit(result);
        APSR.C = carry;
        APSR.V = overflow;
```

Exceptions

None.

A6.7.110 SBFX

Signed Bit Field Extract extracts any number of adjacent bits at any position from one register, sign extends them to 32 bits, and writes the result to the destination register.

Encoding T1

ARMv7-M

```
SBFX<c> <Rd>,<Rn>,#<lsb>,#<width>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	(0)	1	1	0	1	0	0	Rn				0	imm3			Rd				imm2		(0)	widthm1				

```
d = UInt(Rd);  n = UInt(Rn);
lsbit = UInt(imm3:imm2);  widthminus1 = UInt(widthm1);
if d IN {13,15} || n IN {13,15} then UNPREDICTABLE;
```

Assembler syntax

```
SBFX<c><q>  <Rd>, <Rn>, #<lsb>, #<width>
```

where:

`<c><q>` See *Standard assembler syntax fields* on page A6-7.

`<Rd>` Specifies the destination register.

`<Rn>` Specifies the register that contains the first operand.

`<lsb>` is the bit number of the least significant bit in the bitfield, in the range 0-31. This determines the required value of `lsbit`.

`<width>` is the width of the bitfield, in the range 1 to 32-`<lsb>`. The required value of `widthminus1` is `<width>-1`.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    msbit = lsbit + widthminus1;
    if msbit <= 31 then
        R[d] = SignExtend(R[n]<msbit:lsbit>, 32);
    else
        UNPREDICTABLE;
```

Exceptions

None.

A6.7.111 SDIV

Signed Divide divides a 32-bit signed integer register value by a 32-bit signed integer register value, and writes the result to the destination register. The condition code flags are not affected.

Encoding T1 ARMv7-M

`SDIV<c> <Rd>,<Rn>,<Rm>`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	1	1	1	0	0	1	Rn				(1)	(1)	(1)	(1)	Rd				1	1	1	1	Rm			

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d IN {13,15} || n IN {13,15} || m IN {13,15} then UNPREDICTABLE;
```

Assembler syntax

```
SDIV<c><q>  {<Rd>,} <Rn>, <Rm>
```

where:

`<c><q>` See *Standard assembler syntax fields* on page A6-7.

`<Rd>` Specifies the destination register. If `<Rd>` is omitted, this register is the same as `<Rn>`.

`<Rn>` Specifies the register that contains the dividend.

`<Rm>` Specifies the register that contains the divisor.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    if SInt(R[m]) == 0 then
        if IntegerZeroDivideTrappingEnabled() then
            GenerateIntegerZeroDivide();
        else
            result = 0;
    else
        result = RoundTowardsZero(SInt(R[n]) / SInt(R[m]));
    R[d] = result<31:0>;
```

Exceptions

UsageFault.

Notes

Overflow If the signed integer division `0x80000000 / 0xFFFFFFFF` is performed, the pseudocode produces the intermediate integer result $+2^{31}$, which overflows the 32-bit signed integer range. No indication of this overflow case is produced, and the 32-bit result written to `R[d]` is required to be the bottom 32 bits of the binary representation of $+2^{31}$. So the result of the division is `0x80000000`.

A6.7.112 SEV

Send Event is a hint instruction. It causes an event to be signaled to all CPUs within the multiprocessor system. See *Wait For Event and Send Event* on page B1-49 for more details.

This is a NOP-compatible hint, see *NOP-compatible hints* on page A6-16.

Encoding T1 ARMv6-M, ARMv7-M

`SEV<c>`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	1	1	1	0	1	0	0	0	0	0	0

```
// No additional decoding required
```

Encoding T2 ARMv7-M

`SEV<c>.W`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	0	1	0	(1)	(1)	(1)	(1)	1	0	(0)	0	(0)	0	0	0	0	0	0	0	0	1	0	0

```
// No additional decoding required
```

Assembler syntax

`SEV<c><q>`

where:

`<c><q>` See *Standard assembler syntax fields* on page A6-7.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    Hint_SendEvent();
```

Exceptions

None.

A6.7.113 SMLAL

Signed Multiply Accumulate Long multiplies two signed 32-bit values to produce a 64-bit value, and accumulates this with a 64-bit value.

Encoding T1 ARMv7-M

`SMLAL<c> <RdLo>,<RdHi>,<Rn>,<Rm>`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	1	1	1	1	0	0	Rn				RdLo				RdHi				0	0	0	0	Rm			

```
dLo = UInt(RdLo);  dHi = UInt(RdHi);  n = UInt(Rn);  m = UInt(Rm);  setflags = FALSE;
if dLo IN {13,15} || dHi IN {13,15} || n IN {13,15} || m IN {13,15} then UNPREDICTABLE;
if dHi == dLo then UNPREDICTABLE;
```

Assembler syntax

```
SMLAL<c><q>  <RdLo>, <RdHi>, <Rn>, <Rm>
```

where:

`<c><q>` See *Standard assembler syntax fields* on page A6-7.

`<RdLo>` Supplies the lower 32 bits of the accumulate value, and is the destination register for the lower 32 bits of the result.

`<RdHi>` Supplies the upper 32 bits of the accumulate value, and is the destination register for the upper 32 bits of the result.

`<Rn>` Specifies the register that contains the first operand.

`<Rm>` Specifies the register that contains the second operand.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    result = SInt(R[n]) * SInt(R[m]) + SInt(R[dHi]:R[dLo]);
    R[dHi] = result<63:32>;
    R[dLo] = result<31:0>;
```

Exceptions

None.

A6.7.114 SMULL

Signed Multiply Long multiplies two 32-bit signed values to produce a 64-bit result.

Encoding T1 ARMv7-M

`SMULL<c> <RdLo>,<RdHi>,<Rn>,<Rm>`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	1	1	1	0	0	0	Rn				RdLo				RdHi				0	0	0	0	Rm			

```
dLo = UInt(RdLo);  dHi = UInt(RdHi);  n = UInt(Rn);  m = UInt(Rm);  setflags = FALSE;
if dLo IN {13,15} || dHi IN {13,15} || n IN {13,15} || m IN {13,15} then UNPREDICTABLE;
if dHi == dLo then UNPREDICTABLE;
```

Assembler syntax

```
SMULL<c><q>  <RdLo>, <RdHi>, <Rn>, <Rm>
```

where:

`<c><q>` See *Standard assembler syntax fields* on page A6-7.

`<RdLo>` Stores the lower 32 bits of the result.

`<RdHi>` Stores the upper 32 bits of the result.

`<Rn>` Specifies the register that contains the first operand.

`<Rm>` Specifies the register that contains the second operand.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    result = SInt(R[n]) * SInt(R[m]);
    R[dHi] = result<63:32>;
    R[dLo] = result<31:0>;
```

Exceptions

None.

A6.7.115 SSAT

Signed Saturate saturates an optionally-shifted signed value to a selectable signed range.

The Q flag is set if the operation saturates.

Encoding T1 ARMv7-M

`SSAT<c> <Rd>,#<imm5>,<Rn>{,<shift>}`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	(0)	1	1	0	0	sh	0	Rn				0	imm3			Rd				imm2		(0)	sat_imm				

```
if sh == '1' && (imm3:imm2) == '00000' then UNDEFINED;
d = UInt(Rd);  n = UInt(Rn);  saturate_to = UInt(sat_imm)+1;
(shift_t, shift_n) = DecodeImmShift(sh:'0', imm3:imm2);
if d IN {13,15} || n IN {13,15} then UNPREDICTABLE;
```

Assembler syntax

`SSAT<c><q>  <Rd>, #<imm>, <Rn> {,<shift>}`

where:

`<c><q>` See *Standard assembler syntax fields* on page A6-7.

`<Rd>` Specifies the destination register.

`<imm>` Specifies the bit position for saturation, in the range 1 to 32.

`<Rn>` Specifies the register that contains the value to be saturated.

`<shift>` Specifies the optional shift. If `<shift>` is omitted, `LSL #0` is used.

If present, it must be one of:

LSL #*N* *N* must be in the range 0 to 31.

ASR #*N* *N* must be in the range 1 to 31.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    operand = Shift(R[n], shift_t, shift_n, APSR.C);  // APSR.C ignored
    (result, sat) = SignedSatQ(SInt(operand), saturate_to);
    R[d] = SignExtend(result, 32);
    if sat then
        APSR.Q = '1';
```

Exceptions

None.

A6.7.116 STC, STC2

Store Coprocessor stores data from a coprocessor to a sequence of consecutive memory addresses.

If no coprocessor can execute the instruction, a UsageFault exception is generated.

Encoding T1 ARMv7-M

```
STC{L}<c> <coproc>,<CRd>,[<Rn>{,#+/-<imm8>}]
STC{L}<c> <coproc>,<CRd>,[<Rn>,#+/-<imm8>]!
STC{L}<c> <coproc>,<CRd>,[<Rn>],#+/-<imm8>
STC{L}<c> <coproc>,<CRd>,[<Rn>],<option>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	0	P	U	N	W	0	Rn				CRd				coproc				imm8							

```
if P == '0' && U == '0' && D == '0' && W == '0' then UNDEFINED;
if P == '0' && U == '0' && D == '1' && W == '0' then SEE MCRR, MCRR2;
n = UInt(Rn);  cp = UInt(coproc);  imm32 = ZeroExtend(imm8:'00', 32);
index = (P == '1');  add = (U == '1');  wback = (W == '1');
if n == 15 then UNPREDICTABLE;
```

Encoding T2 ARMv7-M

```
STC2{L}<c> <coproc>,<CRd>,[<Rn>{,#+/-<imm8>}]
STC2{L}<c> <coproc>,<CRd>,[<Rn>,#+/-<imm8>]!
STC2{L}<c> <coproc>,<CRd>,[<Rn>],#+/-<imm8>
STC2{L}<c> <coproc>,<CRd>,[<Rn>],<option>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	0	P	U	N	W	0	Rn				CRd				coproc				imm8							

```
if P == '0' && U == '0' && D == '0' && W == '0' then UNDEFINED;
if P == '0' && U == '0' && D == '1' && W == '0' then SEE MCRR, MCRR2;
n = UInt(Rn);  cp = UInt(coproc);  imm32 = ZeroExtend(imm8:'00', 32);
index = (P == '1');  add = (U == '1');  wback = (W == '1');
if n == 15 then UNPREDICTABLE;
```

Assembler syntax

`STC{2}{L}<c><q>  <coproc>,<CRd>,[<Rn>{,#+/-<imm>}]`	Offset. P = 1, W = 0.
`STC{2}{L}<c><q>  <coproc>,<CRd>,[<Rn>,#+/-<imm>]!`	Pre-indexed. P = 1, W = 1.
`STC{2}{L}<c><q>  <coproc>,<CRd>,[<Rn>],#+/-<imm>`	Post-indexed. P = 0, W = 1.
STC{2}{L}<c><q> <coproc>,<CRd>,[<Rn>],<option>	Unindexed. P = 0, W = 0, U = 1.

where:

`2` If specified, selects encoding T2. If omitted, selects encoding T1.

`L` If specified, selects the N == 1 form of the encoding. If omitted, selects the N == 0 form.

`<c><q>` See *Standard assembler syntax fields* on page A6-7.

`<coproc>` Specifies the name of the coprocessor. The standard generic coprocessor names are p0, p1, ..., p15.

`<CRd>` Specifies the coprocessor source register.

`<Rn>` Specifies the base register. This register is allowed to be the SP.

`+/-` Is + or omitted to indicate that the immediate offset is added to the base register value (`add == TRUE`), or – to indicate that the offset is to be subtracted (`add == FALSE`). Different instructions are generated for `#0` and `#-0`.

`<imm>` Specifies the immediate offset added to or subtracted from the value of `<Rn>` to form the address. Allowed values are multiples of 4 in the range 0-1020. For the offset addressing syntax, `<imm>` can be omitted, meaning an offset of 0.

`<option>` Specifies additional instruction options to the coprocessor, as an integer in the range 0-255, surrounded by `{` and `}`. This integer is encoded in the imm8 field of the instruction.

The pre-UAL syntax `STC<c>L` is equivalent to `STCL<c>`.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    if !Coproc_Accepted(cp, ThisInstr()) then
        GenerateCoprocessorException();
    else
        offset_addr = if add then (R[n] + imm32) else (R[n] - imm32);
        address = if index then offset_addr else R[n];
        repeat
            MemA[address,4] = Coproc_GetWordToStore(cp, ThisInstr());  address = address + 4;
        until Coproc_DoneStoring(cp, ThisInstr());
        if wback then R[n] = offset_addr;
```

Exceptions

UsageFault, MemManage, BusFault.

A6.7.117 STM / STMIA / STMEA

Store Multiple Increment After (Store Multiple Empty Ascending) stores multiple registers to consecutive memory locations using an address from a base register. The consecutive memory locations start at this address, and the address just above the last of those locations can optionally be written back to the base register.

Encoding T1 All versions of the Thumb ISA.

`STM<c> <Rn>!,<registers>`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	Rn			register_list							

```
n = UInt(Rn);  registers = '00000000':register_list;  wback = TRUE;
if BitCount(registers) < 1 then UNPREDICTABLE;
```

Encoding T2 ARMv7-M

`STM<c>.W <Rn>{!},<registers>`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	0	0	0	1	0	W	0	Rn				(0)	M	(0)	register_list												

```
n = UInt(Rn);  registers = '0':M:'0':register_list;  wback = (W == '1');
if n == 15 || BitCount(registers) < 2 then UNPREDICTABLE;
if wback && registers<n> == '1' then UNPREDICTABLE;
```

Assembler syntax

`STM<c><q>  <Rn>{!}, <registers>`

where:

`<c><q>` See *Standard assembler syntax fields* on page A6-7.

`<Rn>` The base register. The SP can be used.

`!` Causes the instruction to write a modified value back to `<Rn>`. If `!` is omitted, the instruction does not change `<Rn>`.

`<registers>`

Is a list of one or more registers to be stored, separated by commas and surrounded by `{` and `}`. The lowest-numbered register is stored to the lowest memory address, through to the highest-numbered register to the highest memory address.

Encoding T2 does not support a list containing only one register. If an `STM` instruction with just one register `<Rt>` in the list is assembled to Thumb and encoding T1 is not available, it is assembled to the equivalent `STR<c><q> <Rt>,[<Rn>]{,#4}` instruction.

The SP and PC cannot be in the list.

Encoding T2 is not available for instructions with the base register in the list and ! specified, and the use of such instructions is deprecated. If the base register is not the lowest-numbered register in the list, such an instruction stores an UNKNOWN value for the base register.

STMEA and STMIA are pseudo-instructions for STM, STMEA referring to its use for pushing data onto Empty Ascending stacks.

The pre-UAL syntaxes STM<c>IA and STM<c>EA are equivalent to STM<c>.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    address = R[n];

    for i = 0 to 14
        if registers<i> == '1' then
            if i == n && wback && i != LowestSetBit(registers) then
                MemA[address,4] = bits(32) UNKNOWN;    // encoding T1 only
            else
                MemA[address,4] = R[i];
            address = address + 4;

    if wback then R[n] = R[n] + 4*BitCount(registers);
```

Exceptions

UsageFault, MemManage, BusFault.

A6.7.118 STMDB / STMFD

Store Multiple Decrement Before (Store Multiple Full Descending) stores multiple registers to consecutive memory locations using an address from a base register. The consecutive memory locations end just below this address, and the address of the first of those locations can optionally be written back to the base register.

Encoding T1 ARMv7-M

```
STMDB<c> <Rn>{!},<registers>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	0	0	1	0	0	W	0	Rn				(0)	M	(0)	register_list												

```
if W == '1' && Rn == '1101' then SEE PUSH;
n = UInt(Rn);  registers = '0':M:'0':register_list;  wback = (W == '1');
if n == 15 || BitCount(registers) < 2 then UNPREDICTABLE;
if wback && registers<n> == '1' then UNPREDICTABLE;
```

Assembler syntax

```
STMDB<c><q>  <Rn>{!}, <registers>
```

where:

`<c><q>` See *Standard assembler syntax fields* on page A6-7.

`<Rn>` The base register. If it is the SP and ! is specified, the instruction is treated as described in *PUSH* on page A6-188.

`!` Causes the instruction to write a modified value back to `<Rn>`. Encoded as W = 1.

If `!` is omitted, the instruction does not change `<Rn>`. Encoded as W = 0.

`<registers>`

Is a list of one or more registers to be stored, separated by commas and surrounded by `{` and `}`. The lowest-numbered register is stored to the lowest memory address, through to the highest-numbered register to the highest memory address.

Encoding T1 does not support a list containing only one register. If an `STMDB` instruction with just one register `<Rt>` in the list is assembled to Thumb, it is assembled to the equivalent `STR<c><q> <Rt>,[<Rn>,#-4]{!}` instruction.

The SP and PC cannot be in the list.

`STMFD` is s synonym for `STMDB`, referring to its use for pushing data onto Full Descending stacks.

The pre-UAL syntaxes `STM<c>DB` and `STM<c>FD` are equivalent to `STMDB<c>`.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    address = R[n] - 4*BitCount(registers);

    for i = 0 to 14
        if registers<i> == ’1’ then
            MemA[address,4] = R[i];
            address = address + 4;

    if wback then R[n] = R[n] - 4*BitCount(registers);
```

Exceptions

UsageFault, MemManage, BusFault.

A6.7.119 STR (immediate)

Store Register (immediate) calculates an address from a base register value and an immediate offset, and stores a word from a register to memory. It can use offset, post-indexed, or pre-indexed addressing. See *Memory accesses* on page A6-15 for information about memory accesses.

Encoding T1 All versions of the Thumb ISA.

```
STR<c> <Rt>, [<Rn>{,#<imm5>}]
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	imm5					Rn			Rt		

```
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm5:'00', 32);
index = TRUE;  add = TRUE;  wback = FALSE;
```

Encoding T2 All versions of the Thumb ISA.

```
STR<c> <Rt>,[SP,#<imm8>]
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	1	0	Rt			imm8							

```
t = UInt(Rt);  n = 13;  imm32 = ZeroExtend(imm8:'00', 32);
index = TRUE;  add = TRUE;  wback = FALSE;
```

Encoding T3 ARMv7-M

```
STR<c>.W <Rt>,[<Rn>,#<imm12>]
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	0	1	1	0	0	Rn				Rt				imm12											

```
if Rn == '1111' then UNDEFINED;
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm12, 32);
index = TRUE;  add = TRUE;  wback = FALSE;
if t == 15 then UNPREDICTABLE;
```

Encoding T4 ARMv7-M

```
STR<c> <Rt>,[<Rn>,#-<imm8>]

STR<c> <Rt>,[<Rn>],#+/-<imm8>

STR<c> <Rt>,[<Rn>,#+/-<imm8>]!
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	0	0	1	0	0	Rn				Rt				1	P	U	W	imm8							

```
if P == '1' && U == '1' && W == '0' then SEE STRT;
if Rn == '1101' && P == '1' && U == '0' && W == '1' && imm8 == '00000100' then SEE PUSH;
if Rn == '1111' || (P == '0' && W == '0') then UNDEFINED;
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm8, 32);
index = (P == '1');  add = (U == '1');  wback = (W == '1');
if t == 15 || (wback && n == t) then UNPREDICTABLE;
```

Assembler syntax

`STR<c><q>  <Rt>, [<Rn> {, #+/-<imm>}]`	Offset: `index==TRUE`, `wback==FALSE`
`STR<c><q>  <Rt>, [<Rn>, #+/-<imm>]!`	Pre-indexed: `index==TRUE`, `wback==TRUE`
`STR<c><q>  <Rt>, [<Rn>], #+/-<imm>`	Post-indexed: `index==FALSE`, `wback==TRUE`

where:

`<c><q>` See *Standard assembler syntax fields* on page A6-7.

`<Rt>` Specifies the source register. This register is allowed to be the SP.

`<Rn>` Specifies the base register. This register is allowed to be the SP.

`+/-` Is + or omitted to indicate that the immediate offset is added to the base register value (`add == TRUE`), or – to indicate that the offset is to be subtracted (`add == FALSE`). Different instructions are generated for `#0` and `#-0`.

`<imm>` Specifies the immediate offset added to or subtracted from the value of `<Rn>` to form the address. Allowed values are multiples of 4 in the range 0-124 for encoding T1, multiples of 4 in the range 0-1020 for encoding T2, any value in the range 0-4095 for encoding T3, and any value in the range 0-255 for encoding T4. For the offset addressing syntax, `<imm>` can be omitted, meaning an offset of 0.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    offset_addr = if add then (R[n] + imm32) else (R[n] - imm32);
    address = if index then offset_addr else R[n];
    MemU[address,4] = R[t];
    if wback then R[n] = offset_addr;
```

Exceptions

UsageFault, MemManage, BusFault.

A6.7.120 STR (register)

Store Register (register) calculates an address from a base register value and an offset register value, stores a word from a register to memory. The offset register value can be shifted left by 0, 1, 2, or 3 bits. See *Memory accesses* on page A6-15 for information about memory accesses.

Encoding T1 All versions of the Thumb ISA.

```
STR<c> <Rt>,[<Rn>,<Rm>]
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	0	0	0	Rm			Rn			Rt		

```
t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);
index = TRUE;  add = TRUE;  wback = FALSE;
(shift_t, shift_n) = (SRType_LSL, 0);
```

Encoding T2 ARMv7-M

```
STR<c>.W <Rt>,[<Rn>,<Rm>{,LSL #<imm2>}]
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	0	0	1	0	0	Rn				Rt				0	0	0	0	0	0	imm2		Rm			

```
if Rn == '1111' then UNDEFINED;
t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);
index = TRUE;  add = TRUE;  wback = FALSE;
(shift_t, shift_n) = (SRType_LSL, UInt(imm2));
if t == 15 || m IN {13,15} then UNPREDICTABLE;
```

Assembler syntax

```
STR<c><q>  <Rt>, [<Rn>, <Rm> {, LSL #<shift>}]
```

where:

`<c><q>` See *Standard assembler syntax fields* on page A6-7.

`<Rt>` Specifies the source register. This register is allowed to be the SP.

`<Rn>` Specifies the register that contains the base value. This register is allowed to be the SP.

`<Rm>` Contains the offset that is shifted left and added to the value of `<Rn>` to form the address.

`<shift>` Specifies the number of bits the value from `<Rm>` is shifted left, in the range 0-3. If this option is omitted, a shift by 0 is assumed and both encodings are permitted. If this option is specified, only encoding T2 is permitted.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    offset = Shift(R[m], shift_t, shift_n, APSR.C);
    address = R[n] + offset;
    data = R[t];
    MemU[address,4] = data;
```

Exceptions

UsageFault, MemManage, BusFault.

A6.7.121 STRB (immediate)

Store Register Byte (immediate) calculates an address from a base register value and an immediate offset, and stores a byte from a register to memory. It can use offset, post-indexed, or pre-indexed addressing. See *Memory accesses* on page A6-15 for information about memory accesses.

Encoding T1 All versions of the Thumb ISA.

`STRB<c> <Rt>,[<Rn>,#<imm5>]`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	0	imm5					Rn			Rt		

```
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm5, 32);
index = TRUE;  add = TRUE;  wback = FALSE;
```

Encoding T2 ARMv7-M

`STRB<c>.W <Rt>,[<Rn>,#<imm12>]`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	0	1	0	0	0	Rn				Rt				imm12											

```
if Rn == '1111' then UNDEFINED;
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm12, 32);
index = TRUE;  add = TRUE;  wback = FALSE;
if t IN {13,15} then UNPREDICTABLE;
```

Encoding T3 ARMv7-M

`STRB<c> <Rt>,[<Rn>,#-<imm8>]`

`STRB<c> <Rt>,[<Rn>],#+/-<imm8>`

`STRB<c> <Rt>,[<Rn>,#+/-<imm8>]!`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	0	0	0	0	0	Rn				Rt				1	P	U	W	imm8							

```
if P == '1' && U == '1' && W == '0' then SEE STRBT;
if Rn == '1111' || (P == '0' && W == '0') then UNDEFINED;
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm8, 32);
index = (P == '1');  add = (U == '1');  wback = (W == '1');
if t IN {13,15} || (wback && n == t) then UNPREDICTABLE;
```

Assembler syntax

`STRB<c><q>  <Rt>, [<Rn> {, #+/-<imm>}]`	Offset: `index==TRUE`, `wback==FALSE`
`STRB<c><q>  <Rt>, [<Rn>, #+/-<imm>]!`	Pre-indexed: `index==TRUE`, `wback==TRUE`
`STRB<c><q>  <Rt>, [<Rn>], #+/-<imm>`	Post-indexed: `index==FALSE`, `wback==TRUE`

where:

`<c><q>` See *Standard assembler syntax fields* on page A6-7.

`<Rt>` Specifies the source register.

`<Rn>` Specifies the base register. This register is allowed to be the SP.

`+/-` Is + or omitted to indicate that the immediate offset is added to the base register value (`add == TRUE`), or – to indicate that the offset is to be subtracted (`add == FALSE`). Different instructions are generated for `#0` and `#-0`.

`<imm>` Specifies the immediate offset added to or subtracted from the value of `<Rn>` to form the address. The range of allowed values is 0-31 for encoding T1, 0-4095 for encoding T2, and 0-255 for encoding T3. For the offset addressing syntax, `<imm>` can be omitted, meaning an offset of 0.

The pre-UAL syntax `STR<c>B` is equivalent to `STRB<c>`.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    offset_addr = if add then (R[n] + imm32) else (R[n] - imm32);
    address = if index then offset_addr else R[n];
    MemU[address,1] = R[t]<7:0>;
    if wback then R[n] = offset_addr;
```

Exceptions

MemManage, BusFault.

A6.7.122 STRB (register)

Store Register Byte (register) calculates an address from a base register value and an offset register value, and stores a byte from a register to memory. The offset register value can be shifted left by 0, 1, 2, or 3 bits. See *Memory accesses* on page A6-15 for information about memory accesses.

Encoding T1

All versions of the Thumb ISA.

```
STRB<c> <Rt>,[<Rn>,<Rm>]
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	0	1	0	Rm			Rn			Rt		

```
t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);
index = TRUE;  add = TRUE;  wback = FALSE;
(shift_t, shift_n) = (SRType_LSL, 0);
```

Encoding T2

ARMv7-M

```
STRB<c>.W <Rt>,[<Rn>,<Rm>{,LSL #<imm2>}]
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	0	0	0	0	0	Rn				Rt				0	0	0	0	0	0	imm2		Rm			

```
if Rn == '1111' then UNDEFINED;
t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);
index = TRUE;  add = TRUE;  wback = FALSE;
(shift_t, shift_n) = (SRType_LSL, UInt(imm2));
if t IN {13,15} || m IN {13,15} then UNPREDICTABLE;
```

Assembler syntax

```
STRB<c><q>  <Rt>, [<Rn>, <Rm> {, LSL #<shift>}]
```

where:

`<c><q>` See *Standard assembler syntax fields* on page A6-7.

`<Rn>` Specifies the register that contains the base value. This register is allowed to be the SP.

`<Rm>` Contains the offset that is shifted left and added to the value of `<Rn>` to form the address.

`<shift>` Specifies the number of bits the value from `<Rm>` is shifted left, in the range 0-3. If this option is omitted, a shift by 0 is assumed and both encodings are permitted. If this option is specified, only encoding T2 is permitted.

The pre-UAL syntax `STR<c>B` is equivalent to `STRB<c>`.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    offset = Shift(R[m], shift_t, shift_n, APSR.C);
    address = R[n] + offset;
    MemU[address,1] = R[t]<7:0>;
```

Exceptions

MemManage, BusFault.

A6.7.123 STRBT

Store Register Byte Unprivileged calculates an address from a base register value and an immediate offset, and stores a byte from a register to memory. See *Memory accesses* on page A6-15 for information about memory accesses.

The memory access is restricted as if the processor were running unprivileged. (This makes no difference if the processor is actually running unprivileged.)

Encoding T1 ARMv7-M

`STRBT<c> <Rt>,[<Rn>,#<imm8>]`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	0	0	0	0	0	Rn				Rt				1	1	1	0	imm8							

```
if Rn == '1111' then UNDEFINED;
t = UInt(Rt);  n = UInt(Rn);  postindex = FALSE;  add = TRUE;
register_form = FALSE;  imm32 = ZeroExtend(imm8, 32);
if t IN {13,15} then UNPREDICTABLE;
```

Assembler syntax

```
STRBT<c><q>  <Rt>, [<Rn> {, #<imm>}]
```

where:

`<c><q>` See *Standard assembler syntax fields* on page A6-7.

`<Rt>` Specifies the source register.

`<Rn>` Specifies the base register. This register is allowed to be the SP.

`<imm>` Specifies the immediate offset added to the value of `<Rn>` to form the address. The range of allowed values is 0-255. `<imm>` can be omitted, meaning an offset of 0.

The pre-UAL syntax `STR<c>BT` is equivalent to `STRBT<c>`.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    address = R[n] + imm32;
    MemU_unpriv[address,1] = R[t]<7:0>;
```

Exceptions

MemManage, BusFault.

A6.7.124 STRD (immediate)

Store Register Dual (immediate) calculates an address from a base register value and an immediate offset, and stores two words from two registers to memory. It can use offset, post-indexed, or pre-indexed addressing. See *Memory accesses* on page A6-15 for information about memory accesses.

Encoding T1 ARMv7-M

```
STRD<c> <Rt>,<Rt2>,[<Rn>{,#+/-<imm8>}]
STRD<c> <Rt>,<Rt2>,[<Rn>],#+/-<imm8>
STRD<c> <Rt>,<Rt2>,[<Rn>,#+/-<imm8>]!
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	0	0	P	U	1	W	0	Rn				Rt				Rt2				imm8							

```
if P == '0' && W == '0' then SEE "Related encodings";
t = UInt(Rt);  t2 = UInt(Rt2);  n = UInt(Rn);  imm32 = ZeroExtend(imm8:'00', 32);
index = (P == '1');  add = (U == '1');  wback = (W == '1');
if wback && (n == t || n == t2) then UNPREDICTABLE;
if n == 15 || t IN {13,15} || t2 IN {13,15} then UNPREDICTABLE;
```

Related encodings See *Load/store dual or exclusive, table branch* on page A5-21

Assembler syntax

`STRD<c><q>`	`<Rt>,<Rt2>,[<Rn>{,#+/-<imm>}]`	Offset: `index==TRUE`, `wback==FALSE`
`STRD<c><q>`	`<Rt>,<Rt2>,[<Rn>,#+/-<imm>]!`	Pre-indexed: `index==TRUE`, `wback==TRUE`
`STRD<c><q>`	`<Rt>,<Rt2>,[<Rn>],#+/-<imm>`	Post-indexed: `index==FALSE`, `wback==TRUE`

where:

`<c><q>` See *Standard assembler syntax fields* on page A6-7.

`<Rt>` Specifies the first source register.

`<Rt2>` Specifies the second source register.

`<Rn>` Specifies the base register. This register is allowed to be the SP.

`+/-` Is + or omitted to indicate that the immediate offset is added to the base register value (`add == TRUE`), or – to indicate that the offset is to be subtracted (`add == FALSE`). Different instructions are generated for `#0` and `#-0`.

`<imm>` Specifies the immediate offset added to or subtracted from the value of `<Rn>` to form the address. Allowed values are multiples of 4 in the range 0-1020. For the offset addressing syntax, `<imm>` can be omitted, meaning an offset of 0.

The pre-UAL syntax `STR<c>D` is equivalent to `STRD<c>`.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    offset_addr = if add then (R[n] + imm32) else (R[n] - imm32);
    address = if index then offset_addr else R[n];
    MemA[address,4] = R[t];
    MemA[address+4,4] = R[t2];
    if wback then R[n] = offset_addr;
```

Exceptions

UsageFault, MemManage, BusFault.

A6.7.125 STREX

Store Register Exclusive calculates an address from a base register value and an immediate offset, and stores a word from a register to memory if the executing processor has exclusive access to the memory addressed.

See *Memory accesses* on page A6-15 for information about memory accesses.

Encoding T1 ARMv7-M

```
STREX<c> <Rd>,<Rt>,[<Rn>{,#<imm8>}]
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	0	0	0	0	1	0	0	Rn				Rt				Rd				imm8							

```
d = UInt(Rd);  t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm8:'00', 32);
if d IN {13,15} || t IN {13,15} || n == 15 then UNPREDICTABLE;
if d == n || d == t then UNPREDICTABLE;
```

Assembler syntax

```
STREX<c><q>  <Rd>, <Rt>, [<Rn> {,#<imm>}]
```

where:

`<c><q>` See *Standard assembler syntax fields* on page A6-7.

`<Rd>` Specifies the destination register for the returned status value. The value returned is:

0	if the operation updates memory
1	if the operation fails to update memory.

`<Rt>` Specifies the source register.

`<Rn>` Specifies the base register. This register is allowed to be the SP.

`<imm>` Specifies the immediate offset added to the value of `<Rn>` to form the address. Allowed values are multiples of 4 in the range 0-1020. `<imm>` can be omitted, meaning an offset of 0.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    address = R[n] + imm32;
    if ExclusiveMonitorsPass(address,4) then
        MemA[address,4] = R[t];
        R[d] = 0;
    else
        R[d] = 1;
```

Exceptions

UsageFault, MemManage, BusFault.

A6.7.126 STREXB

Store Register Exclusive Byte derives an address from a base register value, and stores a byte from a register to memory if the executing processor has exclusive access to the memory addressed.

See *Memory accesses* on page A6-15 for information about memory accesses.

Encoding T1 ARMv7-M

`STREXB<c> <Rd>,<Rt>,[<Rn>]`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	0	0	0	1	1	0	0	Rn				Rt				(1)	(1)	(1)	(1)	0	1	0	0	Rd			

```
d = UInt(Rd);  t = UInt(Rt);  n = UInt(Rn);
if d IN {13,15} || t IN {13,15} || n == 15 then UNPREDICTABLE;
if d == n || d == t then UNPREDICTABLE;
```

Assembler syntax

`STREXB<c><q>  <Rd>, <Rt>, [<Rn>]`

where:

`<c><q>` See *Standard assembler syntax fields* on page A6-7.

`<Rd>` Specifies the destination register for the returned status value. The value returned is:

- `0` if the operation updates memory
- `1` if the operation fails to update memory.

`<Rt>` Specifies the source register.

`<Rn>` Specifies the base register. This register is allowed to be the SP.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    address = R[n];
    if ExclusiveMonitorsPass(address,1) then
        MemA[address,1] = R[t];
        R[d] = 0;
    else
        R[d] = 1;
```

Exceptions

MemManage, BusFault.

A6.7.127 STREXH

Store Register Exclusive Halfword derives an address from a base register value, and stores a halfword from a register to memory if the executing processor has exclusive access to the memory addressed.

See *Memory accesses* on page A6-15 for information about memory accesses.

Encoding T1 ARMv7-M

`STREXH<c> <Rd>,<Rt>,[<Rn>]`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	0	0	0	1	1	0	0	Rn				Rt				(1)	(1)	(1)	(1)	0	1	0	1	Rd			

```
d = UInt(Rd);  t = UInt(Rt);  n = UInt(Rn);
if d IN {13,15} || t IN {13,15} || n == 15 then UNPREDICTABLE;
if d == n || d == t then UNPREDICTABLE;
```

Assembler syntax

```
STREXH<c><q>  <Rd>, <Rt>, [<Rn>]
```

where:

`<c><q>` See *Standard assembler syntax fields* on page A6-7.

`<Rd>` Specifies the destination register for the returned status value. The value returned is:

- `0` if the operation updates memory
- `1` if the operation fails to update memory.

`<Rt>` Specifies the source register.

`<Rn>` Specifies the base register. This register is allowed to be the SP.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    address = R[n];
    if ExclusiveMonitorsPass(address,2) then
        MemA[address,2] = R[t];
        R[d] = 0;
    else
        R[d] = 1;
```

Exceptions

UsageFault, MemManage, BusFault.

A6.7.128 STRH (immediate)

Store Register Halfword (immediate) calculates an address from a base register value and an immediate offset, and stores a halfword from a register to memory. It can use offset, post-indexed, or pre-indexed addressing. See *Memory accesses* on page A6-15 for information about memory accesses.

Encoding T1 All versions of the Thumb ISA.

```
STRH<c> <Rt>,[<Rn>{,#<imm5>}]
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	0	0	imm5					Rn			Rt		

```
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm5:'0', 32);
index = TRUE;  add = TRUE;  wback = FALSE;
```

Encoding T2 ARMv7-M

```
STRH<c>.W <Rt>,[<Rn>{,#<imm12>}]
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	0	1	0	1	0	Rn				Rt				imm12											

```
if Rn == '1111' then UNDEFINED;
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm12, 32);
index = TRUE;  add = TRUE;  wback = FALSE;
if t IN {13,15} then UNPREDICTABLE;
```

Encoding T3 ARMv7-M

```
STRH<c> <Rt>,[<Rn>,#-<imm8>]
STRH<c> <Rt>,[<Rn>],#+/-<imm8>
STRH<c> <Rt>,[<Rn>,#+/-<imm8>]!
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	0	0	0	1	0	Rn				Rt				1	P	U	W	imm8							

```
if P == '1' && U == '1' && W == '0' then SEE STRHT;
if Rn == '1111' || (P == '0' && W == '0') then UNDEFINED;
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm8, 32);
index = (P == '1');  add = (U == '1');  wback = (W == '1');
if t IN {13,15} || (wback && n == t) then UNPREDICTABLE;
```

Assembler syntax

`STRH<c><q>  <Rt>, [<Rn> {, #+/-<imm>}]`	Offset: `index==TRUE`, `wback==FALSE`
`STRH<c><q>  <Rt>, [<Rn>, #+/-<imm>]!`	Pre-indexed: `index==TRUE`, `wback==TRUE`
`STRH<c><q>  <Rt>, [<Rn>], #+/-<imm>`	Post-indexed: `index==FALSE`, `wback==TRUE`

where:

`<c><q>` See *Standard assembler syntax fields* on page A6-7.

`<Rt>` Specifies the source register.

`<Rn>` Specifies the base register. This register is allowed to be the SP.

`+/-` Is + or omitted to indicate that the immediate offset is added to the base register value (`add == TRUE`), or – to indicate that the offset is to be subtracted (`add == FALSE`). Different instructions are generated for `#0` and `#-0`.

`<imm>` Specifies the immediate offset added to or subtracted from the value of `<Rn>` to form the address. Allowed values are multiples of 2 in the range 0-62 for encoding T1, any value in the range 0-4095 for encoding T2, and any value in the range 0-255 for encoding T3. For the offset addressing syntax, `<imm>` can be omitted, meaning an offset of 0.

The pre-UAL syntax `STR<c>H` is equivalent to `STRH<c>`.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    offset_addr = if add then (R[n] + imm32) else (R[n] - imm32);
    address = if index then offset_addr else R[n];
    MemU[address,2] = R[t]<15:0>;

    if wback then R[n] = offset_addr;
```

Exceptions

UsageFault, MemManage, BusFault.

A6.7.129 STRH (register)

Store Register Halfword (register) calculates an address from a base register value and an offset register value, and stores a halfword from a register to memory. The offset register value can be shifted left by 0, 1, 2, or 3 bits. See *Memory accesses* on page A6-15 for information about memory accesses.

Encoding T1 All versions of the Thumb ISA.

```
STRH<c> <Rt>,[<Rn>,<Rm>]
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	0	0	1	Rm			Rn			Rt		

```
t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);
index = TRUE;  add = TRUE;  wback = FALSE;
(shift_t, shift_n) = (SRType_LSL, 0);
```

Encoding T2 ARMv7-M

```
STRH<c>.W <Rt>,[<Rn>,<Rm>{,LSL #<imm2>}]
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	0	0	0	1	0	Rn				Rt				0	0	0	0	0	0	imm2		Rm			

```
if Rn == '1111' then UNDEFINED;
t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);
index = TRUE;  add = TRUE;  wback = FALSE;
(shift_t, shift_n) = (SRType_LSL, UInt(imm2));
if t IN {13,15} || m IN {13,15} then UNPREDICTABLE;
```

Assembler syntax

```
STRH<c><q>  <Rt>, [<Rn>, <Rm> {, LSL #<shift>}]
```

where:

`<c><q>` See *Standard assembler syntax fields* on page A6-7.

`<Rn>` Specifies the register that contains the base value. This register is allowed to be the SP.

`<Rm>` Contains the offset that is shifted left and added to the value of `<Rn>` to form the address.

`<shift>` Specifies the number of bits the value from `<Rm>` is shifted left, in the range 0-3. If this option is omitted, a shift by 0 is assumed and both encodings are permitted. If this option is specified, only encoding T2 is permitted.

The pre-UAL syntax `STR<c>H` is equivalent to `STRH<c>`.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    offset = Shift(R[m], shift_t, shift_n, APSR.C);
    address = R[n] + offset;
    MemU[address,2] = R[t]<15:0>;
```

Exceptions

UsageFault, MemManage, BusFault.

A6.7.130 STRHT

Store Register Halfword Unprivileged calculates an address from a base register value and an immediate offset, and stores a halfword from a register to memory. See *Memory accesses* on page A6-15 for information about memory accesses.

The memory access is restricted as if the processor were running unprivileged. (This makes no difference if the processor is actually running unprivileged.)

Encoding T1 ARMv7-M

`STRHT<c> <Rt>,[<Rn>,#<imm8>]`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	0	0	0	1	0	Rn				Rt				1	1	1	0	imm8							

```
if Rn == '1111' then UNDEFINED;
t = UInt(Rt);  n = UInt(Rn);  postindex = FALSE;  add = TRUE;
register_form = FALSE;  imm32 = ZeroExtend(imm8, 32);
if t IN {13,15} then UNPREDICTABLE;
```

Assembler syntax

```
STRHT<c><q>  <Rt>, [<Rn> {, #<imm>}]
```

where:

`<c><q>` See *Standard assembler syntax fields* on page A6-7.

`<Rt>` Specifies the source register.

`<Rn>` Specifies the base register. This register is allowed to be the SP.

`<imm>` Specifies the immediate offset added to the value of `<Rn>` to form the address. The range of allowed values is 0-255. `<imm>` can be omitted, meaning an offset of 0.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    address = R[n] + imm32;
    MemU_unpriv[address,2] = R[t]<15:0>;
```

Exceptions

UsageFault, MemManage, BusFault.

A6.7.131 STRT

Store Register Unprivileged calculates an address from a base register value and an immediate offset, and stores a word from a register to memory. See *Memory accesses* on page A6-15 for information about memory accesses.

The memory access is restricted as if the processor were running unprivileged. (This makes no difference if the processor is actually running unprivileged.)

Encoding T1 ARMv7-M

`STRT<c> <Rt>,[<Rn>,#<imm8>]`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	0	0	1	0	0	Rn				Rt				1	1	1	0	imm8							

```
if Rn == '1111' then UNDEFINED;
t = UInt(Rt);  n = UInt(Rn);  postindex = FALSE;  add = TRUE;
register_form = FALSE;  imm32 = ZeroExtend(imm8, 32);
if t IN {13,15} then UNPREDICTABLE;
```

Assembler syntax

`STRT<c><q>  <Rt>, [<Rn> {, #<imm>}]`

where:

`<c><q>` See *Standard assembler syntax fields* on page A6-7.

`<Rt>` Specifies the source register.

`<Rn>` Specifies the base register. This register is allowed to be the SP.

`<imm>` Specifies the immediate offset added to the value of `<Rn>` to form the address. The range of allowed values is 0-255. `<imm>` can be omitted, meaning an offset of 0.

The pre-UAL syntax `STR<c>T` is equivalent to `STRT<c>`.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    address = R[n] + imm32;
    data = R[t];
    MemU_unpriv[address,4] = data;
```

Exceptions

UsageFault, MemManage, BusFault.

A6.7.132 SUB (immediate)

This instruction subtracts an immediate value from a register value, and writes the result to the destination register. It can optionally update the condition flags based on the result.

Encoding T1 All versions of the Thumb ISA.

`SUBS <Rd>,<Rn>,#<imm3>` Outside IT block.

`SUB<c> <Rd>,<Rn>,#<imm3>` Inside IT block.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	1	1	imm3			Rn			Rd		

```
d = UInt(Rd);  n = UInt(Rn);  setflags = !InITBlock();  imm32 = ZeroExtend(imm3, 32);
```

Encoding T2 All versions of the Thumb ISA.

`SUBS <Rdn>,#<imm8>` Outside IT block.

`SUB<c> <Rdn>,#<imm8>` Inside IT block.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	1	Rdn			imm8							

```
d = UInt(Rdn);  n = UInt(Rdn);  setflags = !InITBlock();  imm32 = ZeroExtend(imm8, 32);
```

Encoding T3 ARMv7-M

`SUB{S}<c>.W <Rd>,<Rn>,#<const>`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	i	0	1	1	0	1	S	Rn				0	imm3			Rd				imm8							

```
if Rd == '1111' && setflags then SEE CMP (immediate);
if Rn == '1101' then SEE SUB (SP minus immediate);
d = UInt(Rd);  n = UInt(Rn);  setflags = (S == '1');  imm32 = ThumbExpandImm(i:imm3:imm8);
if d IN {13,15} || n == 15 then UNPREDICTABLE;
```

Encoding T4 ARMv7-M

`SUBW<c> <Rd>,<Rn>,#<imm12>`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	i	1	0	1	0	1	0	Rn				0	imm3			Rd				imm8							

```
if Rn == '1111' then SEE ADR;
if Rn == '1101' then SEE SUB (SP minus immediate);
d = UInt(Rd);  n = UInt(Rn);  setflags = FALSE;  imm32 = ZeroExtend(i:imm3:imm8, 32);
if d IN {13,15} then UNPREDICTABLE;
```

Assembler syntax

`SUB{S}<c><q>  {<Rd>,} <Rn>, #<const>`	All encodings permitted
`SUBW<c><q>  {<Rd>,} <Rn>, #<const>`	Only encoding T4 permitted

where:

`S` If present, specifies that the instruction updates the flags. Otherwise, the instruction does not update the flags.

`<c><q>` See *Standard assembler syntax fields* on page A6-7.

`<Rd>` Specifies the destination register. If `<Rd>` is omitted, this register is the same as `<Rn>`.

`<Rn>` Specifies the register that contains the first operand. If the SP is specified for `<Rn>`, see *SUB (SP minus immediate)* on page A6-248. If the PC is specified for `<Rn>`, see *ADR* on page A6-30.

`<const>` Specifies the immediate value to be subtracted from the value obtained from `<Rn>`. The range of allowed values is 0-7 for encoding T1, 0-255 for encoding T2 and 0-4095 for encoding T4. See *Modified immediate constants in Thumb instructions* on page A5-15 for the range of allowed values for encoding T3.

When multiple encodings of the same length are available for an instruction, encoding T3 is preferred to encoding T4 (if encoding T4 is required, use the `SUBW` syntax). Encoding T1 is preferred to encoding T2 if `<Rd>` is specified and encoding T2 is preferred to encoding T1 if `<Rd>` is omitted.

The pre-UAL syntax `SUB<c>S` is equivalent to `SUBS<c>`.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    (result, carry, overflow) = AddWithCarry(R[n], NOT(imm32), '1');
    R[d] = result;
    if setflags then
        APSR.N = result<31>;
        APSR.Z = IsZeroBit(result);
        APSR.C = carry;
        APSR.V = overflow;
```

Exceptions

None.

A6.7.133 SUB (register)

This instruction subtracts an optionally-shifted register value from a register value, and writes the result to the destination register. It can optionally update the condition flags based on the result.

Encoding T1 All versions of the Thumb ISA.

`SUBS <Rd>,<Rn>,<Rm>` Outside IT block.

`SUB<c> <Rd>,<Rn>,<Rm>` Inside IT block.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	0	1	Rm			Rn			Rd		

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  setflags = !InITBlock();
(shift_t, shift_n) = (SRType_LSL, 0);
```

Encoding T2 ARMv7-M

`SUB{S}<c>.W <Rd>,<Rn>,<Rm>{,<shift>}`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	0	1	1	1	0	1	S	Rn				(0)	imm3			Rd				imm2		type		Rm			

```
if Rd == '1111' && S == '1' then SEE CMP (register);
if Rn == '1101' then SEE SUB (SP minus register);
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  setflags = (S == '1');
(shift_t, shift_n) = DecodeImmShift(type, imm3:imm2);
if d IN {13,15} || n == 15 || m IN {13,15} then UNPREDICTABLE;
```

Assembler syntax

```
SUB{S}<c><q>  {<Rd>,} <Rn>, <Rm> {,<shift>}
```

where:

S If present, specifies that the instruction updates the flags. Otherwise, the instruction does not update the flags.

<c><q> See *Standard assembler syntax fields* on page A6-7.

<Rd> Specifies the destination register. If <Rd> is omitted, this register is the same as <Rn>.

<Rn> Specifies the register that contains the first operand. If the SP is specified for <Rn>, see *SUB (SP minus register)* on page A6-250.

<Rm> Specifies the register that is optionally shifted and used as the second operand.

<shift> Specifies the shift to apply to the value read from <Rm>. If <shift> is omitted, no shift is applied and both encodings are permitted. If <shift> is specified, only encoding T2 is permitted. The possible shifts and how they are encoded are described in *Shifts applied to a register* on page A6-12.

The pre-UAL syntax SUB<c>S is equivalent to SUBS<c>.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    shifted = Shift(R[m], shift_t, shift_n, APSR.C);
    (result, carry, overflow) = AddWithCarry(R[n], NOT(shifted), ’1’);
    R[d] = result;
    if setflags then
        APSR.N = result<31>;
        APSR.Z = IsZeroBit(result);
        APSR.C = carry;
        APSR.V = overflow;
```

Exceptions

None.

A6.7.134 SUB (SP minus immediate)

This instruction subtracts an immediate value from the SP value, and writes the result to the destination register.

Encoding T1 All versions of the Thumb ISA.

```
SUB<c> SP,SP,#<imm7>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	0	0	0	0	1	imm7						

```
d = 13;  setflags = FALSE;  imm32 = ZeroExtend(imm7:'00', 32);
```

Encoding T2 ARMv7-M

```
SUB{S}<c>.W <Rd>,SP,#<const>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	i	0	1	1	0	1	S	1	1	0	1	0	imm3			Rd				imm8							

```
if Rd == '1111' && S == '1' then SEE CMP (immediate);
d = UInt(Rd);  setflags = (S == '1');  imm32 = ThumbExpandImm(i:imm3:imm8);
if d == 15 then UNPREDICTABLE;
```

Encoding T3 ARMv7-M

```
SUBW<c> <Rd>,SP,#<imm12>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	i	1	0	1	0	1	0	1	1	0	1	0	imm3			Rd				imm8							

```
d = UInt(Rd);  setflags = FALSE;  imm32 = ZeroExtend(i:imm3:imm8, 32);
if d == 15 then UNPREDICTABLE;
```

Assembler syntax

`SUB{S}<c><q>  {<Rd>,} SP, #<const>`	All encodings permitted
`SUBW<c><q>  {<Rd>,} SP, #<const>`	Only encoding T4 permitted

where:

`S` If present, specifies that the instruction updates the flags. Otherwise, the instruction does not update the flags.

`<c><q>` See *Standard assembler syntax fields* on page A6-7.

`<Rd>` Specifies the destination register. If `<Rd>` is omitted, this register is SP.

`<const>` Specifies the immediate value to be added to the value obtained from SP. Allowed values are multiples of 4 in the range 0-508 for encoding T1 and any value in the range 0-4095 for encoding T3. See *Modified immediate constants in Thumb instructions* on page A5-15 for the range of allowed values for encoding T2.

When both 32-bit encodings are available for an instruction, encoding T2 is preferred to encoding T3 (if encoding T3 is required, use the `SUBW` syntax).

The pre-UAL syntax `SUB<c>S` is equivalent to `SUBS<c>`.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    (result, carry, overflow) = AddWithCarry(SP, NOT(imm32), '1');
    R[d] = result;
    if setflags then
        APSR.N = result<31>;
        APSR.Z = IsZeroBit(result);
        APSR.C = carry;
        APSR.V = overflow;
```

Exceptions

None.

A6.7.135 SUB (SP minus register)

This instruction subtracts an optionally-shifted register value from the SP value, and writes the result to the destination register.

Encoding T1 All versions of the Thumb ISA.

```
SUB{S}<c> <Rd>,SP,<Rm>{,<shift>}
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	0	1	1	1	0	1	S	1	1	0	1	(0)	imm3			Rd				imm2		type		Rm			

```
d = UInt(Rd);  m = UInt(Rm);  setflags = (S == '1');
(shift_t, shift_n) = DecodeImmShift(type, imm3:imm2);
if d == 13 && (shift_t != SRType_LSL || shift_n > 3) then UNPREDICTABLE;
if d == 15 || m IN {13,15} then UNPREDICTABLE;
```

Assembler syntax

```
SUB{S}<c><q>  {<Rd>,} SP, <Rm> {,<shift>}
```

where:

`S` If present, specifies that the instruction updates the flags. Otherwise, the instruction does not update the flags.

`<c><q>` See *Standard assembler syntax fields* on page A6-7.

`<Rd>` Specifies the destination register. If `<Rd>` is omitted, this register is SP.

`<Rm>` Specifies the register that is optionally shifted and used as the second operand.

`<shift>` Specifies the shift to apply to the value read from `<Rm>`. If `<shift>` is omitted, no shift is applied. The possible shifts and how they are encoded are described in *Shifts applied to a register* on page A6-12.

If `<Rd>` is SP or omitted, `<shift>` is only permitted to be `LSL #0`, `LSL #1`, `LSL #2` or `LSL #3`.

The pre-UAL syntax `SUB<c>S` is equivalent to `SUBS<c>`.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    shifted = Shift(R[m], shift_t, shift_n, APSR.C);
    (result, carry, overflow) = AddWithCarry(SP, NOT(shifted), '1');
    R[d] = result;
    if setflags then
        APSR.N = result<31>;
        APSR.Z = IsZeroBit(result);
        APSR.C = carry;
        APSR.V = overflow;
```

Exceptions

None.

A6.7.136 SVC (formerly SWI)

Generates a supervisor call. See *Exceptions* in the *ARM Architecture Reference Manual*.

Use it as a call to an operating system to provide a service.

Encoding T1

All versions of the Thumb ISA.

```
SVC<c> #<imm8>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	1	1	1	1	imm8							

```
imm32 = ZeroExtend(imm8, 32);
// imm32 is for assembly/disassembly, and is ignored by hardware. SVC handlers in some
// systems interpret imm8 in software, for example to determine the required service.
```

Assembler syntax

```
SVC<c><q>  #<imm>
```

where:

`<c><q>` See *Standard assembler syntax fields* on page A6-7.

`<imm>` Specifies an 8-bit immediate constant.

The pre-UAL syntax `SWI<c>` is equivalent to `SVC<c>`.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CallSupervisor();
```

Exceptions

SVCall.

A6.7.137 SXTB

Signed Extend Byte extracts an 8-bit value from a register, sign extends it to 32 bits, and writes the result to the destination register. You can specify a rotation by 0, 8, 16, or 24 bits before extracting the 8-bit value.

Encoding T1 ARMv6-M, ARMv7-M

`SXTB<c> <Rd>,<Rm>`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	0	0	1	0	0	1	Rm			Rd		

```
d = UInt(Rd);  m = UInt(Rm);  rotation = 0;
```

Encoding T2 ARMv7-M

`SXTB<c>.W <Rd>,<Rm>{,<rotation>}`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	1	0	0	1	0	0	1	1	1	1	1	1	1	1	Rd				1	(0)	rotate		Rm			

```
d = UInt(Rd);  m = UInt(Rm);  rotation = UInt(rotate:'000');
if d IN {13,15} || m IN {13,15} then UNPREDICTABLE;
```

Assembler syntax

```
SXTB<c><q>  <Rd>, <Rm> {, <rotation>}
```

where:

`<c><q>` See *Standard assembler syntax fields* on page A6-7.

`<Rd>` Specifies the destination register.

`<Rm>` Specifies the register that contains the operand.

`<rotation>`

This can be any one of:

- `ROR #8`.
- `ROR #16`.
- `ROR #24`.
- Omitted.

Note

If your assembler accepts shifts by `#0` and treats them as equivalent to no shift or `LSL #0`, then it must accept `ROR #0` here. It is equivalent to omitting `<rotation>`.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    rotated = ROR(R[m], rotation);
    R[d] = SignExtend(rotated<7:0>, 32);
```

Exceptions

None.

A6.7.138 SXTH

Signed Extend Halfword extracts a 16-bit value from a register, sign extends it to 32 bits, and writes the result to the destination register. You can specify a rotation by 0, 8, 16, or 24 bits before extracting the 16-bit value.

Encoding T1 ARMv6-M, ARMv7-M

```
SXTH<c> <Rd>,<Rm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	0	0	1	0	0	0	Rm			Rd		

```
d = UInt(Rd);  m = UInt(Rm);  rotation = 0;
```

Encoding T2 ARMv7-M

```
SXTH<c>.W <Rd>,<Rm>{,<rotation>}
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	1	0	0	0	0	0	1	1	1	1	1	1	1	1	Rd				1	(0)	rotate		Rm			

```
d = UInt(Rd);  m = UInt(Rm);  rotation = UInt(rotate:'000');
if d IN {13,15} || m IN {13,15} then UNPREDICTABLE;
```

Assembler syntax

```
SXTH<c><q>  <Rd>, <Rm> {, <rotation>}
```

where:

`<c><q>` See *Standard assembler syntax fields* on page A6-7.

`<Rd>` Specifies the destination register.

`<Rm>` Specifies the register that contains the operand.

`<rotation>`

This can be any one of:

- `ROR #8`.
- `ROR #16`.
- `ROR #24`.
- Omitted.

Note

If your assembler accepts shifts by `#0` and treats them as equivalent to no shift or `LSL #0`, then it must accept `ROR #0` here. It is equivalent to omitting `<rotation>`.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    rotated = ROR(R[m], rotation);
    R[d] = SignExtend(rotated<15:0>, 32);
```

Exceptions

None.

A6.7.139 TBB, TBH

Table Branch Byte causes a PC-relative forward branch using a table of single byte offsets. A base register provides a pointer to the table, and a second register supplies an index into the table. The branch length is twice the value of the byte returned from the table.

Table Branch Halfword causes a PC-relative forward branch using a table of single halfword offsets. A base register provides a pointer to the table, and a second register supplies an index into the table. The branch length is twice the value of the halfword returned from the table.

Encoding T1 ARMv7-M

`TBB<c> [<Rn>,<Rm>]` Outside or last in IT block

`TBH<c> [<Rn>,<Rm>,LSL #1]` Outside or last in IT block

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	0	0	0	1	1	0	1	Rn				(1)	(1)	(1)	(1)	(0)	(0)	(0)	(0)	0	0	0	H	Rm			

```
n = UInt(Rn);  m = UInt(Rm);  is_tbh = (H == '1');
if n == 13 || m IN {13,15} then UNPREDICTABLE;
if InITBlock() && !LastInITBlock() then UNPREDICTABLE;
```

Assembler syntax

```
TBB<c><q>  [<Rn>, <Rm>]
```

```
TBH<c><q>  [<Rn>, <Rm>, LSL #1]
```

where:

`<c><q>` See *Standard assembler syntax fields* on page A6-7.

`<Rn>` The base register. This contains the address of the table of branch lengths. This register can be the PC. If it is, the table immediately follows this instruction.

`<Rm>` The index register.

For `TBB`, this contains an integer pointing to a single byte in the table. The offset in the table is the value of the index.

For `TBH`, this contains an integer pointing to a halfword in the table. The offset in the table is twice the value of the index.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    if is_tbh then
        halfwords = UInt(MemU[R[n]+LSL(R[m],1), 2]);
    else
        halfwords = UInt(MemU[R[n]+R[m], 1]);
    BranchWritePC(PC + 2*halfwords);
```

Exceptions

MemManage, BusFault.

A6.7.140 TEQ (immediate)

Test Equivalence (immediate) performs an exclusive OR operation on a register value and an immediate value. It updates the condition flags based on the result, and discards the result.

Encoding T1   ARMv7-M

```
TEQ<c> <Rn>,#<const>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	i	0	0	1	0	0	1	Rn				0	imm3			1	1	1	1	imm8							

```
n = UInt(Rn);
(imm32, carry) = ThumbExpandImm_C(i:imm3:imm8, APSR.C);
if n IN {13,15} then UNPREDICTABLE;
```

Assembler syntax

```
TEQ<c><q>  <Rn>, #<const>
```

where:

`<c><q>` See *Standard assembler syntax fields* on page A6-7.

`<Rn>` The register that contains the operand.

`<const>` The immediate value to be tested against the value obtained from `<Rn>`. See *Modified immediate constants in Thumb instructions* on page A5-15 for the range of allowed values.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    result = R[n] EOR imm32;
    APSR.N = result<31>;
    APSR.Z = IsZeroBit(result);
    APSR.C = carry;
    // APSR.V unchanged
```

Exceptions

None.

A6.7.141 TEQ (register)

Test Equivalence (register) performs an exclusive OR operation on a register value and an optionally-shifted register value. It updates the condition flags based on the result, and discards the result.

Encoding T1 ARMv7-M

`TEQ<c> <Rn>,<Rm>{,<shift>}`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	0	1	0	1	0	0	1	Rn				(0)	imm3			1	1	1	1	imm2		type		Rm			

```
n = UInt(Rn);  m = UInt(Rm);
(shift_t, shift_n) = DecodeImmShift(type, imm3:imm2);
if n IN {13,15} || m IN {13,15} then UNPREDICTABLE;
```

Assembler syntax

```
TEQ<c><q>  <Rn>, <Rm> {,<shift>}
```

where:

`<c><q>` See *Standard assembler syntax fields* on page A6-7.

`<Rn>` Specifies the register that contains the first operand.

`<Rm>` Specifies the register that is optionally shifted and used as the second operand.

`<shift>` Specifies the shift to apply to the value read from `<Rm>`. If `<shift>` is omitted, no shift is applied. The possible shifts and how they are encoded are described in *Shifts applied to a register* on page A6-12.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    (shifted, carry) = Shift_C(R[m], shift_t, shift_n, APSR.C);
    result = R[n] EOR shifted;
    APSR.N = result<31>;
    APSR.Z = IsZeroBit(result);
    APSR.C = carry;
    // APSR.V unchanged
```

Exceptions

None.

A6.7.142 TST (immediate)

Test (immediate) performs a logical AND operation on a register value and an immediate value. It updates the condition flags based on the result, and discards the result.

Encoding T1 ARMv7-M

```
TST<c> <Rn>,#<const>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	i	0	0	0	0	0	1	Rn				0	imm3			1	1	1	1	imm8							

```
n = UInt(Rn);
(imm32, carry) = ThumbExpandImm_C(i:imm3:imm8, APSR.C);
if n IN {13,15} then UNPREDICTABLE;
```

Assembler syntax

```
TST<c><q>  <Rn>, #<const>
```

where:

`<c><q>` See *Standard assembler syntax fields* on page A6-7.

`<Rn>` Specifies the register that contains the operand.

`<const>` Specifies the immediate value to be tested against the value obtained from `<Rn>`. See *Modified immediate constants in Thumb instructions* on page A5-15 for the range of allowed values.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    result = R[n] AND imm32;
    APSR.N = result<31>;
    APSR.Z = IsZeroBit(result);
    APSR.C = carry;
    // APSR.V unchanged
```

Exceptions

None.

A6.7.143 TST (register)

Test (register) performs a logical AND operation on a register value and an optionally-shifted register value. It updates the condition flags based on the result, and discards the result.

Encoding T1 All versions of the Thumb ISA.

```
TST<c> <Rn>,<Rm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	0	1	0	0	0	Rm			Rn		

```
n = UInt(Rdn);  m = UInt(Rm);
(shift_t, shift_n) = (SRType_LSL, 0);
```

Encoding T2 ARMv7-M

```
TST<c>.W  <Rn>,<Rm>{,<shift>}
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	0	1	0	0	0	0	1	Rn				(0)	imm3			1	1	1	1	imm2		type		Rm			

```
n = UInt(Rn);  m = UInt(Rm);
(shift_t, shift_n) = DecodeImmShift(type, imm3:imm2);
if n IN {13,15} || m IN {13,15} then UNPREDICTABLE;
```

Assembler syntax

```
TST<c><q>  <Rn>, <Rm> {,<shift>}
```

where:

`<c><q>` See *Standard assembler syntax fields* on page A6-7.

`<Rn>` Specifies the register that contains the first operand.

`<Rm>` Specifies the register that is optionally shifted and used as the second operand.

`<shift>` Specifies the shift to apply to the value read from `<Rm>`. If `<shift>` is omitted, no shift is applied and both encodings are permitted. If `<shift>` is specified, only encoding T2 is permitted. The possible shifts and how they are encoded are described in *Shifts applied to a register* on page A6-12.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    (shifted, carry) = Shift_C(R[m], shift_t, shift_n, APSR.C);
    result = R[n] AND shifted;
    APSR.N = result<31>;
    APSR.Z = IsZeroBit(result);
    APSR.C = carry;
    // APSR.V unchanged
```

Exceptions

None.

A6.7.144 UBFX

Unsigned Bit Field Extract extracts any number of adjacent bits at any position from one register, zero extends them to 32 bits, and writes the result to the destination register.

Encoding T1 ARMv7-M

`UBFX<c> <Rd>,<Rn>,#<lsb>,#<width>`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	(0)	1	1	1	1	0	0	Rn				0	imm3			Rd				imm2		(0)	widthm1				

```
d = UInt(Rd);  n = UInt(Rn);
lsbit = UInt(imm3:imm2);  widthminus1 = UInt(widthm1);
if d IN {13,15} || n IN {13,15} then UNPREDICTABLE;
```

Assembler syntax

```
UBFX<c><q>  <Rd>, <Rn>, #<lsb>, #<width>
```

where:

`<c><q>` See *Standard assembler syntax fields* on page A6-7.

`<Rd>` Specifies the destination register.

`<Rn>` Specifies the register that contains the first operand.

`<lsb>` is the bit number of the least significant bit in the bitfield, in the range 0-31. This determines the required value of `lsbit`.

`<width>` is the width of the bitfield, in the range 1 to 32-`<lsb>`). The required value of `widthminus1` is `<width>`-1.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    msbit = lsbit + widthminus1;
    if msbit <= 31 then
        R[d] = ZeroExtend(R[n]<msbit:lsbit>, 32);
    else
        UNPREDICTABLE;
```

Exceptions

None.

A6.7.145 UDIV

Unsigned Divide divides a 32-bit unsigned integer register value by a 32-bit unsigned integer register value, and writes the result to the destination register. The condition code flags are not affected.

Encoding T1 ARMv7-M

`UDIV<c> <Rd>,<Rn>,<Rm>`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	1	1	1	0	1	1	Rn				(1)	(1)	(1)	(1)	Rd				1	1	1	1	Rm			

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d IN {13,15} || n IN {13,15} || m IN {13,15} then UNPREDICTABLE;
```

Assembler syntax

```
UDIV<c><q>  {<Rd>,} <Rn>, <Rm>
```

where:

`<c><q>` See *Standard assembler syntax fields* on page A6-7.

`<Rd>` Specifies the destination register. If `<Rd>` is omitted, this register is the same as `<Rn>`.

`<Rn>` Specifies the register that contains the dividend.

`<Rm>` Specifies the register that contains the divisor.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    if UInt(R[m]) == 0 then
        if IntegerZeroDivideTrappingEnabled() then
            GenerateIntegerZeroDivide();
        else
            result = 0;
    else
        result = RoundTowardsZero(UInt(R[n]) / UInt(R[m]));
    R[d] = result<31:0>;
```

Exceptions

UsageFault.

A6.7.146 UMLAL

Unsigned Multiply Accumulate Long multiplies two unsigned 32-bit values to produce a 64-bit value, and accumulates this with a 64-bit value.

Encoding T1 ARMv7-M

```
UMLAL<c> <RdLo>,<RdHi>,<Rn>,<Rm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	1	1	1	1	1	0	Rn				RdLo				RdHi				0	0	0	0	Rm			

```
dLo = UInt(RdLo);  dHi = UInt(RdHi);  n = UInt(Rn);  m = UInt(Rm);  setflags = FALSE;
if dLo IN {13,15} || dHi IN {13,15} || n IN {13,15} || m IN {13,15} then UNPREDICTABLE;
if dHi == dLo then UNPREDICTABLE;
```

Assembler syntax

```
UMLAL<c><q>  <RdLo>, <RdHi>, <Rn>, <Rm>
```

where:

`<c><q>` See *Standard assembler syntax fields* on page A6-7.

`<RdLo>` Supplies the lower 32 bits of the accumulate value, and is the destination register for the lower 32 bits of the result.

`<RdHi>` Supplies the upper 32 bits of the accumulate value, and is the destination register for the upper 32 bits of the result.

`<Rn>` Specifies the register that contains the first operand.

`<Rm>` Specifies the register that contains the second operand.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    result = UInt(R[n]) * UInt(R[m]) + UInt(R[dHi]:R[dLo]);
    R[dHi] = result<63:32>;
    R[dLo] = result<31:0>;
```

Exceptions

None.

A6.7.147 UMULL

Unsigned Multiply Long multiplies two 32-bit unsigned values to produce a 64-bit result.

Encoding T1 ARMv7-M

```
UMULL<c> <RdLo>,<RdHi>,<Rn>,<Rm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	1	1	1	0	1	0	Rn				RdLo				RdHi				0	0	0	0	Rm			

```
dLo = UInt(RdLo);  dHi = UInt(RdHi);  n = UInt(Rn);  m = UInt(Rm);  setflags = FALSE;
if dLo IN {13,15} || dHi IN {13,15} || n IN {13,15} || m IN {13,15} then UNPREDICTABLE;
if dHi == dLo then UNPREDICTABLE;
```

Assembler syntax

```
UMULL<c><q>  <RdLo>, <RdHi>, <Rn>, <Rm>
```

where:

`<c><q>`	See *Standard assembler syntax fields* on page A6-7.
`<RdLo>`	Stores the lower 32 bits of the result.
`<RdHi>`	Stores the upper 32 bits of the result.
`<Rn>`	Specifies the register that contains the first operand.
`<Rm>`	Specifies the register that contains the second operand.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    result = UInt(R[n]) * UInt(R[m]);
    R[dHi] = result<63:32>;
    R[dLo] = result<31:0>;
```

Exceptions

None.

A6.7.148 USAT

Unsigned Saturate saturates an optionally-shifted signed value to a selected unsigned range.

The Q flag is set if the operation saturates.

Encoding T1

ARMv7-M

`USAT<c> <Rd>,#<imm5>,<Rn>{,<shift>}`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	(0)	1	1	1	0	sh	0	Rn				0	imm3			Rd				imm2		(0)	sat_imm				

```
if sh == '1' && (imm3:imm2) == '00000' then UNDEFINED;
d = UInt(Rd);  n = UInt(Rn);  saturate_to = UInt(sat_imm);
(shift_t, shift_n) = DecodeImmShift(sh:'0', imm3:imm2);
if d IN {13,15} || n IN {13,15} then UNPREDICTABLE;
```

Assembler syntax

```
USAT<c><q>  <Rd>, #<imm>, <Rn> {,<shift>}
```

where:

`<c><q>` See *Standard assembler syntax fields* on page A6-7.

`<Rd>` Specifies the destination register.

`<imm>` Specifies the bit position for saturation, in the range 0 to 31.

`<Rn>` Specifies the register that contains the value to be saturated.

`<shift>` Specifies the optional shift. If present, it must be one of:

LSL #*N* *N* must be in the range 0 to 31.

ASR #*N* *N* must be in the range 1 to 31.

If `<shift>` is omitted, `LSL #0` is used.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    operand = Shift(R[n], shift_t, shift_n, APSR.C);  // APSR.C ignored
    (result, sat) = UnsignedSatQ(SInt(operand), saturate_to);
    R[d] = ZeroExtend(result, 32);
    if sat then
        APSR.Q = '1';
```

Exceptions

None.

A6.7.149 UXTB

Unsigned Extend Byte extracts an 8-bit value from a register, zero extends it to 32 bits, and writes the result to the destination register. You can specify a rotation by 0, 8, 16, or 24 bits before extracting the 8-bit value.

Encoding T1 ARMv6-M, ARMv7

```
UXTB<c> <Rd>,<Rm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	0	0	1	0	1	1	Rm			Rd		

```
d = UInt(Rd);  m = UInt(Rm);  rotation = 0;
```

Encoding T2 ARMv7-M

```
UXTB<c>.W <Rd>,<Rm>{,<rotation>}
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	1	0	0	1	0	1	1	1	1	1	1	1	1	1	Rd				1	(0)	rotate		Rm			

```
d = UInt(Rd);  m = UInt(Rm);  rotation = UInt(rotate:'000');
if d IN {13,15} || m IN {13,15} then UNPREDICTABLE;
```

Assembler syntax

```
UXTB<c><q>  <Rd>, <Rm> {, <rotation>}
```

where:

`<c><q>` See *Standard assembler syntax fields* on page A6-7.

`<Rd>` Specifies the destination register.

`<Rm>` Specifies the register that contains the second operand.

`<rotation>`

This can be any one of:

- `ROR #8`.
- `ROR #16`.
- `ROR #24`.
- Omitted.

Note

If your assembler accepts shifts by `#0` and treats them as equivalent to no shift or `LSL #0`, then it must accept `ROR #0` here. It is equivalent to omitting `<rotation>`.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    rotated = ROR(R[m], rotation);
    R[d] = ZeroExtend(rotated<7:0>, 32);
```

Exceptions

None.

A6.7.150 UXTH

Unsigned Extend Halfword extracts a 16-bit value from a register, zero extends it to 32 bits, and writes the result to the destination register. You can specify a rotation by 0, 8, 16, or 24 bits before extracting the 16-bit value.

Encoding T1 ARMv6-M, ARMv7

```
UXTH<c> <Rd>,<Rm>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	0	0	1	0	1	0	Rm			Rd		

```
d = UInt(Rd);  m = UInt(Rm);  rotation = 0;
```

Encoding T2 ARMv7-M

```
UXTH<c>.W <Rd>,<Rm>{,<rotation>}
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	1	0	0	0	0	1	1	1	1	1	1	1	1	1	Rd				1	(0)	rotate		Rm			

```
d = UInt(Rd);  m = UInt(Rm);  rotation = UInt(rotate:'000');
if d IN {13,15} || m IN {13,15} then UNPREDICTABLE;
```

Assembler syntax

```
UXTH<c><q>  <Rd>, <Rm> {, <rotation>}
```

where:

`<c><q>` See *Standard assembler syntax fields* on page A6-7.

`<Rd>` Specifies the destination register.

`<Rm>` Specifies the register that contains the second operand.

`<rotation>`

This can be any one of:

- `ROR #8`.
- `ROR #16`.
- `ROR #24`.
- Omitted.

Note

If your assembler accepts shifts by `#0` and treats them as equivalent to no shift or `LSL #0`, then it must accept `ROR #0` here. It is equivalent to omitting `<rotation>`.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    rotated = ROR(R[m], rotation);
    R[d] = ZeroExtend(rotated<15:0>, 32);
```

Exceptions

None.

A6.7.151 WFE

Wait For Event is a hint instruction. If the Event Register is clear, it suspends execution in the lowest power state available consistent with a fast wakeup without the need for software restoration, until a reset, exception or other event occurs. See *Wait For Event and Send Event* on page B1-49 for more details.

For general hint behavior, see *NOP-compatible hints* on page A6-16.

Encoding T1 ARMv6-M, ARMv7-M

`WFE<c>`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	1	1	1	0	0	1	0	0	0	0	0

```
// No additional decoding required
```

Encoding T2 ARMv7-M

`WFE<c>.W`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	0	1	0	(1)	(1)	(1)	(1)	1	0	(0)	0	(0)	0	0	0	0	0	0	0	0	0	1	0

```
// No additional decoding required
```

Assembler syntax

`WFE<c><q>`

where:

`<c><q>` See *Standard assembler syntax fields* on page A6-7.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    if EventRegistered() then
        ClearEventRegister();
    else
        WaitForEvent();
```

Exceptions

None.

A6.7.152 WFI

Wait For Interrupt is a hint instruction. It suspends execution, in the lowest power state available consistent with a fast wakeup without the need for software restoration, until a reset, asynchronous exception or other event occurs. See *Wait For Interrupt* on page B1-51 for more details.

For general hint behavior, see *NOP-compatible hints* on page A6-16.

Encoding T1 ARMv6-M, ARMv7-M

`WFI<c>`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	1	1	1	0	0	1	1	0	0	0	0

```
// No additional decoding required
```

Encoding T2 ARMv7-M

`WFI<c>.W`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	0	1	0	(1)	(1)	(1)	(1)	1	0	(0)	0	(0)	0	0	0	0	0	0	0	0	0	1	1

```
// No additional decoding required
```

Assembler syntax

`WFI<c><q>`

where:

`<c><q>` See *Standard assembler syntax fields* on page A6-7.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    WaitForInterrupt();
```

Exceptions

None.

Notes

PRIMASK ~~If PRIMASK is set and FAULTMASK is clear, an asynchronous exception that has a higher group priority than any active exception and a higher group priority than BASEPRI results in a `WFI` instruction exit. If the group priority of the exception is less than or equal to the execution group priority, the exception is ignored.~~

A6.7.153 YIELD

`YIELD` is a hint instruction. It allows software with a multithreading capability to indicate to the hardware that it is performing a task, for example a spinlock, that could be swapped out to improve overall system performance. Hardware can use this hint to suspend and resume multiple code threads if it supports the capability.

For general hint behavior, see *NOP-compatible hints* on page A6-16.

Encoding T1 ARMv6-M, ARMv7-M

`YIELD<c>`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	1	1	1	0	0	0	1	0	0	0	0

```
// No additional decoding required
```

Encoding T2 ARMv7-M

`YIELD<c>.W`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	0	1	0	(1)	(1)	(1)	(1)	1	0	(0)	0	(0)	0	0	0	0	0	0	0	0	0	0	1

```
// No additional decoding required
```

Assembler syntax

`YIELD<c><q>`

where:

`<c><q>` See *Standard assembler syntax fields* on page A6-7.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    Hint_Yield();
```

Exceptions

None.

Part B

System Level Architecture

Chapter B1
System Level Programmers' Model

This chapter contains information on the system programmers' model. It covers the registers, exception model and fault handling capabilities. The chapter is made up of the following sections:

B1.1 Introduction to the system level

The ARM architecture is defined in a hierarchical manner, where the features are described in Chapter A2 *Application Level Programmers' Model* at the application level, with underlying system support. What features are available and how they are supported is defined in the architecture profiles, making the system level support profile specific. Deprecated features can be found in an appendix to this manual. See page AppxD-1.

As stated in *Privileged execution* on page A2-13, programs can execute in a privileged or unprivileged manner. System level support requires privileged access, allowing it the access permissions to configure and control the resources. This is typically supported by an operating system, which provides system services to the applications, either transparently, or through application initiated service calls. The operating system is also responsible for servicing interrupts and other system events, making exceptions a key component of the system level programmers' model.

In addition, ARMv7-M is a departure from the normal architecture evolution in that it has been designed to take the ARM architecture to lower cost/performance points than previously supported as well as having a strong migration path to ARMv7-R and the broad spectrum of embedded processing.

Note

In deeply embedded systems, particularly at low cost/performance points, the distinction between the operating system and application is sometimes blurred, resulting in the software developed as a homogeneous codebase.

B1.2 ARMv7-M: a memory mapped architecture

ARMv7-M is a memory-mapped architecture, meaning physical addresses as well as processor registers are architecturally assigned to provide event entry points (vectors), system control and configuration. Exception handler entry points are maintained in a table of address pointers.

The address space `0xE0000000` to `0xFFFFFFFF` is reserved for system level use. The first 1MB of the system address space (`0xE0000000` to `0xE00FFFFF`) is reserved by ARM and known as the Private Peripheral Bus (PPB), with the rest of the address space (from `0xE0100000`) IMPLEMENTATION DEFINED with some memory attribute restrictions. See *The system address map* on page B3-2 for more details.

Within the PPB address space, a 4kB block in the range `0xE000E000` to `0xE000EFFF` is assigned for system control and known as the System Control Space (SCS). The SCS supports:

- CPU ID registers
- General control and configuration (including the vector table base address)
- System handler support (for system interrupts and exceptions)
- A SysTick system timer
- A Nested Vectored Interrupt Controller (NVIC), supporting up to 496 discrete external interrupts. All exceptions and interrupts share a common prioritization model
- Fault status and control registers
- The Protected Memory System Architecture (PMSAv7)
- Processor debug

See *System Control Space (SCS)* on page B3-6 for more details.

B1.3 System level operation and terminology overview

Several concepts are critical to the understanding of the system level architecture support.

B1.3.1 Modes, Privilege and Stacks

Mode, privilege and stack pointer are key concepts used in ARMv7-M.

Mode The microcontroller profile supports two modes (Thread and Handler modes). Handler mode is entered as a result of an exception. An exception return can only be issued in Handler mode.

Thread mode is entered on Reset, and can be entered as a result of an exception return.

Privilege Code can execute as privileged or unprivileged. Unprivileged execution limits or excludes access to some resources. Privileged execution has access to all resources. Handler mode is always privileged. Thread mode can be privileged or unprivileged.

Stack Pointer Two separate banked stack pointers exist, the Main Stack Pointer, and the Process Stack pointer. The Main Stack Pointer can be used in either Thread or Handler mode. The Process Stack Pointer can only be used in Thread mode. See *The SP registers* on page B1-8 for more details.

Table B1-1 shows the relationship between mode, privilege and stack pointer usage.

Table B1-1 Mode, privilege and stack relationship

Mode	Privilege	Stack Pointer	Example (typical) usage model
Handler	Privileged	Main	Exception handling
Handler	Unprivileged	Any	Reserved combination (Handler is always privileged)
Handler	Any	Process	Reserved combination (Handler always uses the Main stack)
Thread	Privileged	Main	Execution of a privileged process/thread using a common stack in a system that only supports privileged access

Table B1-1 Mode, privilege and stack relationship (continued)

Mode	Privilege	Stack Pointer	Example (typical) usage model
Thread	Privileged	Process	Execution of a privileged process/thread using a stack reserved for that process/thread in a system that only supports privileged access, or where a mix of privileged and unprivileged threads exist.
Thread	Unprivileged	Main	Execution of an unprivileged process/thread using a common stack in a system that supports privileged and unprivileged (User) access
Thread	Unprivileged	Process	Execution of an unprivileged process/thread using a stack reserved for that process/thread in a system that supports privileged and unprivileged (User) access

B1.3.2 Exceptions

An exception is a condition that changes the normal flow of control in a program. Exception behavior splits into two parts:

- Exception recognition when an exception event is generated and presented to the processor
- Exception processing (activation) when the processor is executing an exception entry, exception return, or exception handler code sequence. Migration from exception recognition to processing can be instantaneous.

Exceptions can be split into four categories

Reset Reset is a special form of exception which terminates current execution in a potentially unrecoverable way when reset is asserted. When reset is de-asserted execution is restarted from a fixed point.

Supervisor call (SVCall)
An exception which is explicitly caused by the SVC instruction. A supervisor call is used by application code to make a system (service) call to an underlying operating system. The SVC instruction enables the application to issue a system call that requires privileged access to the system and will execute in program order with respect to the application. ARMv7-M also supports an interrupt driven service calling mechanism PendSV (see *Interrupts* in *Overview of the exceptions supported* on page B1-14 for more details).

Fault A fault is an exception which results from an error condition due to instruction execution. Faults can be reported synchronously or asynchronously to the instruction which caused them. In general, faults are reported synchronously. The Imprecise BusFault is an asynchronous fault supported in the ARMv7-M profile.

A synchronous fault is always reported with the instruction which caused the fault. An asynchronous fault does not guarantee how it is reported with respect to the instruction which caused the fault.

Synchronous debug monitor exceptions are classified as faults. Watchpoints are asynchronous and treated as an interrupt.

Interrupt An interrupt is an exception, other than a reset, fault or a supervisor call. All interrupts are asynchronous to the instruction stream. Typically interrupts are used by other elements within the system which wish to communicate with the processor, including software running on other processors.

Each exception has:

- a priority level
- an exception number
- a vector in memory which defines the entry-point (address) for execution on taking the exception. The associated code is described as the exception handler or the interrupt service routine (ISR).

Each synchronous exception, other than reset, is in one of three possible states:

- an Inactive exception is one which is not Pending or Active
- a Pending exception is one where the exception event has been generated, and which has not yet started being processed on the processor
- an Active exception is one whose handler has been started on a processor, but processing is not complete. An Active exception can be either running or pre-empted by a higher priority exception.

Asynchronous exceptions can be in one of the three possible states or both Pending and Active at the same time, where one instance of the exception is Active, and a second instance of the exception is Pending.

Priority Levels and Execution Pre-emption

All exceptions are assigned a priority level, the exception priority. Three exceptions have fixed values, while all others can be altered by privileged software. In addition, the instruction stream executing on the processor has a priority level associated with it, the execution priority. An exception whose exception priority is sufficiently[1] higher than the execution priority will become active. In this case, the currently running instruction stream is pre-empted, and the exception that is taken is activated.

When an instruction stream is pre-empted by an exception other than reset, key context information is saved onto the stack automatically. Execution branches to the code pointed to by the exception vector that has been activated.

[1] The concept of sufficiently higher relates to priority grouping within the exception prioritization model. Priority grouping is explained in *Priority grouping* on page B1-18

The execution priority can be boosted by software using registers provided for this purpose, otherwise it is the highest priority of all the exceptions that are active. See *Execution priority and priority boosting within the core* on page B1-18 for more details.

Exceptions can occur during the exception activation, for example as a result of a memory fault while pushing context information. Late-arrival exception optimizations are permissible. The behavior of these cases is described in *Exceptions on exception entry* on page B1-33.

Exception Return

When in handler mode, an exception handler can return. If the exception is both Active and Pending (a second instance of the exception has occurred while it is being serviced), it is re-entered or becomes Pending according to the prioritization rules. If the exception is Active only, it becomes Inactive. The key information that was stacked is restored, and execution returns to the code pre-empted by the exception. The target of the exception return is determined by the Exception Return Link, a value stored in the link register on exception entry.

On an exception return, there can be a pending exception which is of sufficiently high priority that the pending exception will pre-empt the execution being returned to. This will result in an exception entry sequence immediately after an exception return sequence. This condition is referred to as chaining of the exceptions. Hardware can optimize chaining of exceptions to remove the need to restore and re-save the key context state; this optimization is referred to as Tail-chaining. See *Tail-chaining and exceptions on exception return* on page B1-35 for details.

Faults can occur during the exception return, for example as a result of a memory fault while popping previous state off the stack. The behavior in this and other cases is explained in *Derived exceptions* on page B1-34.

B1.3.3 Execution State

ARMv7-M only executes Thumb instructions, both 16-bit and 32-bit instructions as described in *The ARM Architecture – M profile* on page A1-2, and hence is always executing in Thumb state. Thumb state is indicated by an execution status bit (EPSR.T == 1) within the architecture, see *The special-purpose program status registers (xPSR)* on page B1-8. ARMv7-M is consistent with the software programming model and interworking support of additional execution states in other profiles. Setting EPSR.T to zero in ARMv7-M causes a fault when the next instruction executes, because all instructions in this state are ~~undefined~~.

B1.3.4 Debug State

Debug state is entered when a core is configured to halt on a debug event, and a debug event occurs. See Chapter C1 *ARMv7-M Debug* for more details.

The alternative debug mechanism (generate a DebugMonitor exception) does not use Debug state.

B1.4 Registers

The ARMv7-M profile has the following registers closely coupled to the core:

- general purpose registers R0-R12
- 2 Stack Pointer registers, SP_main and SP_process (banked versions of R13)
- the Link Register, LR (R14)
- the Program Counter, PC
- status registers for flags, exception/interrupt level, and execution state bits
- mask registers associated with managing the prioritization scheme for exceptions and interrupts
- a control register (CONTROL) to identify the current stack and thread mode privilege level.

All other registers described in this specification are memory mapped.

Note

Register access restrictions where stated apply to normal execution. Debug restrictions can differ, see *General rules applying to debug register access* on page C1-6, *Debug Core Register Selector Register (DCRSR)* on page C1-22 and *Debug Core Register Data Register (DCRDR)* on page C1-23.

B1.4.1 The SP registers

There are two stacks supported in ARMv7-M, each with its own (banked) stack pointer register.

- the Main stack – SP_main
- the Process stack – SP_process.

ARMv7-M implementations treat bits [1:0] as RAZ/WI. Software should treat bits [1:0] as SBZP for maximum portability across ARMv7 profiles.

The SP that is used by instructions which explicitly reference the SP is selected according to the function `LookUpSP()` described in *Pseudocode details for ARM core register access in the Thumb instruction set* on page B1-12.

The stack pointer that is used in exception entry and exit is described in the pseudocode sequences of the exception entry and exit, see *Exception entry behavior* on page B1-21 and *Exception return behavior* on page B1-25 for more details. SP_main is selected and initialized on reset, see *Reset behavior* on page B1-20.

B1.4.2 The special-purpose program status registers (xPSR)

Program status at the system level breaks down into three categories. They can be accessed as individual registers, a combination of any two from three, or a combination of all three using the `MRS` and `MSR` instructions.

- The Application Program Status Register APSR - User writeable flags. APSR handling of user writeable flags by the `MSR` and `MRS` instructions is consistent across all ARMv7 profiles.

- The Interrupt Program Status Register IPSR – Exception Number (for current execution)

- The Execution Program Status Register EPSR - Execution state bits

The APSR, IPSR and EPSR registers are allocated as mutually exclusive bitfields within a 32-bit register. The combination of the APSR, IPSR and EPSR registers is referred to as the xPSR register.

Table B1-2 The xPSR register layout

	31	30	29	28	27	26 25	24	23 16	15 10	9	8 0
APSR	N	Z	C	V	Q						
IPSR											0 or Exception Number
EPSR						ICI/IT	T		ICI/IT	[a]	

a. While EPSR[9] is reserved, its associated bit location in memory for stacking xPSR context information is allocated to stack alignment support, see *Stack alignment on exception entry* on page B1-24

The APSR is modified by flag setting instructions and used to evaluate conditional execution in IT and conditional branch instructions. The flags (NZCVQ) are as described in *ARM core registers* on page A2-11. The flags are UNPREDICTABLE on reset.

The IPSR is written on exception entry and exit. It can be read using an `MRS` instruction. Writes to the IPSR by an `MSR` instruction are ignored. The IPSR Exception Number field is defined as follows:

- When in Thread mode, the value is 0.
- When in Handler mode, the value reflects the exception number as defined in *Exception number definition* on page B1-16.

The exception number is used to determine the currently executing exception and its entry vector (see *Exception number definition* on page B1-16 and *The vector table* on page B1-16).

On reset, the core is in Thread mode and the Exception Number field of the IPSR is cleared. As a result, the value 1 (the Reset Exception Number) is a transitory value, and not a valid IPSR Exception Number.

The EPSR contains the T-bit and overlaid IT/ICI execution state bits to support the `IT` instruction or interrupt-continue load/store instructions. All fields read as zero using an `MRS` instruction. `MSR` writes are ignored.

The EPSR T-bit supports the ARM architecture interworking model, however, as ARMv7-M only supports execution of Thumb instructions, it must always be maintained with the value T-bit == 1. Updates to the PC which comply with the Thumb instruction interworking rules must update the T-bit accordingly. The execution of an instruction with the EPSR T-bit clear will cause an invalid state (INVSTATE) UsageFault. The T-bit is set and the IT/ICI bits cleared on reset (see *Reset behavior* on page B1-20 for details).

The ICI/IT bits are used for saved IT state or saved exception-continuable instruction state.

- The IT bits provide context information for the conditional execution of a sequence of instructions such that it can be interrupted and restarted at the appropriate point. See the `IT` instruction definition in Chapter A6 *Thumb Instruction Details* for more information.
- The ICI bits provide information on the outstanding register list for exception-continuable multi-cycle load and store instructions.

The IT/ICI bits are assigned according to Table B1-3.

Table B1-3 ICI/IT bit allocation in the EPSR

EPSR[26:25]	EPSR[15:12]	EPSR[11:10]	Additional Information
IT[1:0]	IT[7:4]	IT[3:2]	See *ITSTATE* on page A6-10.
ICI[7:6] ('00')	ICI[5:2] (reg_num)	ICI[1:0] ('00')	See *Exceptions in LDM and STM operations* on page B1-30.

The IT feature takes precedence over the ICI feature if an exception-continuable instruction is used within an IT construct. In this situation, the multi-cycle load or store instruction is treated as restartable.

All unused bits in the individual or combined registers are reserved.

B1.4.3 The special-purpose mask registers

There are three special-purpose registers which are used for the purpose of priority boosting. Their function is explained in detail in *Execution priority and priority boosting within the core* on page B1-18:

- the exception mask register (PRIMASK) which has a 1-bit value
- the base priority mask (BASEPRI) which has an 8-bit value
- the fault mask (FAULTMASK) which has a 1-bit value.

All mask registers are cleared on reset. All unprivileged writes are ignored.

The formats of the mask registers are illustrated in Table B1-4.

Table B1-4 The special-purpose mask registers

	31 … 8	7 … 1	0
PRIMASK	RESERVED		PM
FAULTMASK	RESERVED		FM
BASEPRI	RESERVED	BASEPRI	

Implementations can support an IMPLEMENTATION DEFINED number of priorities in powers of 2. Where fewer than 256 priorities are implemented, the low-order bits of the BASEPRI field corresponding to the unimplemented priority bits are RAZ/WI.

These registers can be accessed using the MSR/MRS instructions. The MSR instruction includes an additional register mask value BASEPRI_MAX, which updates BASEPRI only where the new value increases the priority level (decreases BASEPRI to a non-zero value). See *MSR (register)* on page B4-8 for details.

In addition:

- FAULTMASK is set by the execution of the instruction: CPSID f
- FAULTMASK is cleared by the execution of the instruction: CPSIE f
- PRIMASK is set by the execution of the instruction: CPSID i
- PRIMASK is cleared by the execution of the instruction: CPSIE i.

B1.4.4 The special-purpose ~~control~~ register

The special-purpose CONTROL register is a 2-bit register defined as follows:

- bit [0] defines the Thread mode privilege (Handler mode is always privileged)
  - 0: Thread mode has privileged access
  - 1: Thread mode has unprivileged access.
- bit [1] defines the stack to be used
  - 0: SP_main is used as the current stack
  - 1: For Thread mode, SP_process is used for the current stack. For Handler mode, this value is reserved.
  - Software can update bit [1] in Thread mode. Explicit writes from Handler mode are ignored.
  - The bit is updated on exception entry and exception return. See the pseudocode in *Exception entry behavior* on page B1-21 and *Exception return behavior* on page B1-25 for more details.
- bits [31:2] reserved.

The CONTROL register is cleared on reset. The MRS instruction is used to read the register, and the MSR instruction is used to write the register. Unprivileged write accesses are ignored.

An ISB barrier instruction is required to ensure a CONTROL register write access takes effect before the next instruction is executed.

B1.4.5 Reserved special-purpose register bits

All unused bits in special-purpose registers are reserved. MRS and MSR instructions that access reserved bits treat them as RAZ/WI. For future software compatibility, the bits are UNK/SBZP. Software should write them to zero when initializing the register for a new process, otherwise software should restore reserved bits when updating or restoring a special-purpose register.

B1.4.6 Special-purpose register updates and the memory order model

With the exception of writes to the CONTROL register, all changes to special-purpose registers from a CPS or MSR instruction are guaranteed:

- not to affect those instructions or preceding instructions in program order
- to be visible to all instructions that appear in program order after those changes.

B1.4.7 Register related definitions for pseudocode

Two register types are used in the system programmers' model pseudocode:

- 32- bit core registers
- 32-bit memory mapped registers.

See Appendix I *Register Index* for a list of ARMv7-M registers.

For bit fields associated with registers the convention adopted is to describe them as <register_name>.<bitfield_name> or by a specific bit reference. For example:

- AIRCR.SYSRESETREQ
- CONTROL [1]

Pseudocode details for ARM core register access in the Thumb instruction set

The following pseudocode supports access to the general-purpose registers for the Thumb instruction set operations defined in *Alphabetical list of ARMv7-M Thumb instructions* on page A6-17:

```
// The M-profile execution modes.

enumeration Mode {Mode_Thread, Mode_Handler};

// The names of the core registers. SP is a banked register.

enumeration RName {RName0, RName1, RName2, RName3, RName4, RName5, RName6,
                   RName7, RName8, RName9, RName10, RName11, RName12,
                   RNameSP_main, RNameSP_process, RName_LR, RName_PC};

// The physical array of core registers.
//
// _R[RName_PC] is defined to be the address of the current instruction.
// The offset of 4 bytes is applied to it by the register access functions.

array bits(32) _R[RName];

// LookUpSP()
// ==========

RName LookUpSP()
    RName SP;
    Mode CurrentMode;
    if CONTROL<1> == 1
        if CurrentMode==Mode_Thread then
            SP is RNameSP_process;
        else
            UNPREDICTABLE;
    else
        SP is RNameSP_main;
    return SP;


// R[] - non-assignment form
// =========================

bits(32) R[integer n]
    assert n >= 0 && n <= 15;
    if n == 15 then
        result = _R[RName_PC] + 4;
    elsif n == 14 then
        result = _R[RName_LR]
    elsif n == 13 then
        LookUpSP();
        result = _R[SP];
    else
        result = _R[RName:n];
    return result;


// R[] - assignment form
// =====================

R[integer n] = bits(32) value
    assert n >= 0 && n <= 14;
    if n == 13 then
        LookUpSP();
        _R[SP] = value;
    else
        _R[RName:n] = value;
    return;


// BranchTo()
// ==========

BranchTo(bits(32) address)
    _R[RName_PC] = address;
    return;
```

B1.5 Exception model

The exception model is central to the architecture and system correctness in the ARMv7-M profile. The ARMv7-M profile differs from the other ARMv7 profiles in using hardware saving and restoring of key context state on exception entry and exit, and using a table of vectors to determine the exception entry points. In addition, the exception categorization in the ARMv7-M profile is different from the other ARMv7 profiles.

B1.5.1 Overview of the exceptions supported

The following exceptions are supported by the ARMv7-M profile.

Reset Two levels of reset are supported by the ARMv7-M profile. The levels of reset control which register bit fields are forced to their reset values on the de-assertion of reset.

- Power-On Reset (POR) resets the core, System Control Space and debug logic.
- Local Reset resets the core and System Control Space except some fault and debug-related resources. For more details, see *Debug and reset* on page C1-13.

The Reset exception is permanently enabled, and has a fixed priority of -3.

NMI – Non Maskable Interrupt Non Maskable Interrupt is the highest priority exception other than reset. It is permanently enabled and has a fixed priority of -2.

NMI can be set to the Pending state by software (see *Interrupt Control State Register (ICSR)* on page B3-12) or hardware.

HardFault HardFault is the generic fault that exists for all classes of fault that cannot be handled by any of the other exception mechanisms. HardFault will typically be used for unrecoverable system failure situations, though this is not required, and some uses of HardFault might be recoverable. HardFault is permanently enabled and has a fixed priority of -1.

HardFault is used for fault escalation, see *Priority escalation* on page B1-19 for details.

MemManage The MemManage fault handles memory protection related faults which are determined by the Memory Protection Unit or by fixed memory protection constraints, for both instruction and data generated memory transactions. The fault can be disabled (in this case, a MemManage fault will escalate to HardFault). MemManage has a configurable priority.

BusFault The BusFault fault handles memory related faults other than those handled by the MemManage fault for both instruction and data generated memory transactions. Typically these faults will arise from errors detected on the system buses. Implementations are permitted to report synchronous or asynchronous BusFaults according to the circumstances that trigger the exceptions. The fault can be disabled (in this case, a BusFault will escalate to HardFault). BusFault has a configurable priority.

UsageFault The UsageFault fault handles non-memory related faults caused by the instruction execution. A number of different situations will cause usage faults, including:

- UNDEFINED Instructions
- invalid state on instruction execution

- errors on exception return
- disabled or unavailable coprocessor access.

The following can cause usage faults when the core is configured to report them:

- unaligned addresses on word and halfword memory accesses
- division by zero.

UsageFault can be disabled (in this case, a UsageFault will escalate to HardFault). UsageFault has a configurable priority.

Debug Monitor In general, a DebugMonitor exception is a synchronous exception and classified as a fault. Watchpoints are asynchronous and behave as an interrupt. Debug monitor exceptions occur when halting debug is disabled, and debug monitor support is enabled. DebugMonitor has a configurable priority. See *Priority escalation* on page B1-19 and *Debug event behavior* on page C1-14 for more details.

SVCall This supervisor call handles the exception caused by the SVC instruction. SVCall is permanently enabled and has a configurable priority.

Interrupts The ARMv7-M profile supports two system level interrupts – PendSV for software generation of asynchronous system calls, and SysTick for a Timer integral to the ARMv7-M profile – along with up to 496 external interrupts. All interrupts have a configurable priority.

PendSV[1] is a permanently enabled interrupt, controlled using ICSR.PENDSVSET and ICSR.PENDSVCLR (see *Interrupt Control State Register (ICSR)* on page B3-12). SysTick can not be disabled.

Note

While hardware generation of a SysTick event can be suppressed, ICSR.PENDSTSET and ICSR.PENDSTCLR (see *Interrupt Control State Register (ICSR)* on page B3-12) are always available to software.

All other interrupts can be disabled. Interrupts can be set to or cleared from the Pending state by software, and interrupts other than PendSV can be set to the Pending state by hardware.

See *Fault behavior* on page B1-39 for a definitive list of all the possible causes of faults, the type of fault reported, and the fault status register bits used to identify the faults.

[1] A service (system) call is used by an application which requires a service from an underlying operating system. The service call associated with PendSV executes when the interrupt is taken. For a service call which executes synchronously with respect to program execution use the SVC instruction (the SVCall exception).

B1.5.2 Exception number definition

All exceptions have an associated exception number as defined in Table B1-5.

Table B1-5 Exception numbers

Exception number	Exception
1	Reset
2	NMI
3	HardFault
4	MemManage
5	BusFault
6	UsageFault
7-10	RESERVED
11	SVCall
12	Debug Monitor
13	RESERVED
14	PendSV
15	SysTick
16	External Interrupt(0)
…	…
16 + N	External Interrupt(N)

B1.5.3 The vector table

The vector table contains the initialization value for the stack pointer on reset, and the entry point addresses for all exception handlers. The exception number (see above) defines the order of entries in the vector table associated with exception handler entry as illustrated in Table B1-6.

Table B1-6 Vector table format

word offset	Description – all pointer address values
0	SP_main (reset value of the Main stack pointer)
Exception Number	Exception using that Exception Number

On reset (power-up and local reset, see *Overview of the exceptions supported* on page B1-14 and *Reset management* on page B1-47), the vector table is initialized to an IMPLEMENTATION DEFINED value in the CODE or SRAM partition of the ARMv7-M memory map. The table's current location can be determined or relocated using the Vector Table Offset Register (VTOR), see *Vector Table Offset Register (VTOR)* on page B3-13.

The Vector table must be naturally aligned to a power of two whose alignment value is greater than or equal to (Number of Exceptions supported x 4), with a minimum alignment of 128 bytes.The entry at offset 0 is used to initialize the value for SP_main, see *The SP registers* on page B1-8. All other entries must have bit [0] set, as the bit is used to define the EPSR T-bit on exception entry (see *Reset behavior* on page B1-20 and *Exception entry behavior* on page B1-21 for details).

On exception entry, if bit [0] of the associated vector table entry is clear, execution of the first instruction will cause an INVSTATE UsageFault (see *The special-purpose program status registers (xPSR)* on page B1-8 and *Fault behavior* on page B1-39, Table B1-9 on page B1-40). On reset, this will escalate to a HardFault (see *Priority escalation* on page B1-19) due to the UsageFault being disabled on reset.

B1.5.4 Exception priorities and pre-emption

The priority algorithm treats lower numbers as taking higher precedence, that is, the lower the assigned value the higher the priority level. Exceptions assigned the same priority level adopt a fixed priority order for selection within the architecture according to their exception number.

Reset, non-maskable interrupts (NMI) and HardFault execute at fixed priorities of -3, -2, and -1 respectively. All other exception priorities can be set under software control and are cleared on reset.

The priorities of all exceptions are set in registers within the System Control Space (specifically, registers within the system control block and NVIC).

When multiple exceptions have the same priority number, the pending exception with the lowest exception number takes precedence. Once an exception is active, only exceptions with a higher priority (lower priority number) can pre-empt it.

The priority field is an 8-bit field. In systems supporting less than 256 priority levels, the most significant bits are used to define the priority. This aligns the use of the priority field with the priority grouping mechanism described below.

If the priority of an exception is changed when it is pending or active, this change is guaranteed to take effect on completion of a subsequent `DSB` instruction.

Priority grouping

Exception priority is split into two parts, the group priority and the sub-priority. The allocation of bits to the two parts is controlled by the PRIGROUP field of the Application Interrupt and Reset Control Register (AIRCR). This is a 3-bit field which indicates how many of the most significant bits within the 8-bit priority field for a given exception number are allocated to priority grouping as illustrated in Table B1-7.

Table B1-7 Priority grouping

	Exception Priority Field [7:0]	
PRIGROUP value	**Priority Group bit field**	**Sub-priority bit field**
0	[7:1]	[0]
1	[7:2]	[1:0]
2	[7:3]	[2:0]
3	[7:4]	[3:0]
4	[7:5]	[4:0]
5	[7:6]	[5:0]
6	[7]	[6:0]
7	-	[7:0]

The priority group bit field is used to define the priority for pre-emption. The priority associated with this field is referred to as the group priority. Where multiple pending exceptions share the same group priority, the sub-priority bit field is then used to resolve the priority within a group. Where two pending exceptions have the same priority the lower pending exception number has priority over the higher pending exception number.

The group priorities of Reset, NMI and HardFault are -3, -2, and -1 respectively, regardless of the value of PRIGROUP.

Execution priority and priority boosting within the core

The execution priority is defined to be the highest priority formulated from a set of values:

- the priorities of all Active exceptions (lowest priority number)
- the impact of PRIMASK, FAULTMASK and BASEPRI values (see below)

Note

Note: If the priority of an active exception is changed it can affect the execution priority. Therefore, the execution priority can be different from the priority of the running exception. This ensures that dynamic priority management avoids priority inversion from a new exception with respect to the active exception stack.

The priority can be boosted by the following mechanisms:

- PRIMASK: setting this mask bit raises the execution priority to 0. This prevents all exceptions with configurable priority from activating, other than through the fault escalation mechanism (see *Priority escalation*). This also has a special impact on `WFI` (see *WFI* on page A6-277).
- FAULTMASK: setting this mask bit raises the execution priority to -1. FAULTMASK can only be set when the execution priority is not NMI or HardFault (the priority value is greater than or equal to zero). Setting ~~the FaultMask~~ raises the priority of the exception handler to the level of a HardFault. FAULTMASK is cleared automatically on all exception returns except a return from NMI.
- BASEPRI: can be written with a value from N (lowest configurable priority) to 1. When this register is cleared to 0, it has no effect on the ~~current~~ priority. A non-zero value will act as a priority mask~~,~~ ~~affecting~~ the execution priority when the priority defined by BASEPRI is higher than the current executing priority.

The priority boosting mechanisms only affect the group priority. They have no effect on the sub-priority. The sub-priority is only used to sort pending exception priorities, and does not affect active exceptions.

Priority escalation

When the currently executing group priority is less than HardFault, the priority of exceptions is *escalated* to HardFault in the following cases:

- when the group priority of a pending synchronous fault or supervisor call is lower than or equal to the currently executing group priority, inhibiting normal pre-emption. This applies to all synchronous exceptions (a fault or SVCall) including the `BKPT` instruction but excluding all other DebugMonitor faults.
- on an occurence of a configurable fault that is not enabled.

A fault which is escalated to a HardFault retains the ReturnAddress() behavior of the original fault. See the pseudocode definition of ReturnAddress() in *Exception entry behavior* on page B1-21 for more details. For the behavior of the affected exceptions occuring when the currently executing group priority is that of a HardFault or higher, see *Unrecoverable exception cases* on page B1-44.

The following situations are examples of pending exceptions which give rise to priority escalation:

- a configurable system exception handler causes the same kind of exception as is being serviced. For example, an ~~undefined~~ instruction is encountered in a UsageFault handler.
- a configurable fault handler generates a different fault and the handler for it is the same or lower priority

- the configurable fault is not enabled
- an SVC instruction when PRIMASK is set to 1.

Note

Enabled interrupts are not escalated – they are set to the Pending state. Disabled interrupts are ignored.

Asynchronous faults (Imprecise BusFaults) are set to the Pending state and are entered according to normal priority rules when enabled. They are treated as HardFault exceptions when disabled.

SVCall, PendSV and critical region code avoidance

Context switching typically requires a critical region of code where interrupts must be disabled to avoid context corruption of key data structures during the change. This can be a severe constraint on system design and deterministic performance. ARMv7-M can support context switching with no critical region such that interrupts never need to be disabled.

An example usage model supporting critical region avoidance is to configure both SVCall and PendSV with the same, lowest exception priority. SVCall can be used for supervisor calls from threads, and PendSV can be used to handle context critical work offloaded from the exception handlers, including the equal priority SVCall handler. Because SVCall and PendSV have the same execution priority they will never pre-empt each other, therefore one will always process to completion before the other starts. SVCall and PendSV exceptions are always enabled, which means they will each execute at some point once all other exceptions have been handled. In addition, the associated exception handlers do not need to check whether they are returning to a process on exit with this usage model, as the PendSV exception will occur when returning to a process.

The example has all context switch requests issued by setting PendSV to Pending, however, both SVCall and PendSV exceptions can be used for context switching because they do not interfere with each other. While not the only usage model, support of critical region software avoidance is a key feature of ARMv7-M, specifically the support provided by the SVCall and PendSV exception specifications.

B1.5.5 Reset behavior

The assertion of reset causes the current execution state to be abandoned without being saved. On the de-assertion of reset, all registers controlled by the reset assertion contain their reset values, and the following actions are performed.

For global declarations see *Register related definitions for pseudocode* on page B1-12.
For helper functions and procedures see *Miscellaneous helper procedures and functions* on page AppxG-22.

```
// TakeReset()
// ============

~~integer NestedActivation;           /* used for Handler => Thread check when value == 1 */~~
bit ExceptionActive[*];             /* conceptual array of 1-bit values for all exceptions */
bits(32) vectortable = '00':VTOR<29:7>:'0000000';
Mode CurrentMode;
```

```
TakeReset()
    R[0..12] = bits(32) UNKNOWN;
    SP_main = MemA[vectortable,4] & 0xFFFFFFFC;
    SP_process = ((bits(30) UNKNOWN):'00');
    LR = 0xFFFFFFFF;                /* preset to an illegal exception return value */
    tmp = MemA[vectortable+4,4]
    PC = tmp AND 0xFFFFFFFE;        /* address of reset service routine */
    tbit = tmp<0>;
    CurrentMode = Mode_Thread;
    APSR = bits(32) UNKNOWN;        /* flags UNPREDICTABLE from reset */
    IPSR<8:0> = 0x0;                /* Exception Number cleared */
    EPSR.T = tbit;                  /* T bit set from vector */
    EPSR.IT<7:0> = 0x0;             /* IT/ICI bits cleared */
    PRIMASK<0> = '0';               /* priority mask cleared at reset */
    FAULTMASK<0> = '0';             /* fault mask cleared at reset */
    BASEPRI<7:0> = 0x0;             /* base priority disabled at reset */
    CONTROL<1:0> = '00';            /* current stack is Main, thread is privileged */
    ResetSCSRegs();                 /* catch-all function for System Control Space reset */
    ~~NestedActivation = 0x0;~~         ~~/* initialised value for base thread */~~
    ExceptionActive[*] = '0';       /* all exceptions Inactive */
    ClearExclusiveLocal();          /* Synchronization (LDREX*/STREX*) monitor support */
                                    /* to open access state. */
    ClearEventRegister()              /* see WFE instruction for more details */
```

ExceptionActive[*] is a conceptual array of active flag bits for all exceptions (fixed priority system exceptions, configurable priority system exceptions, and external interrupts). The fixed priority active flags are conceptual only, and are not required to exist in a system register.

B1.5.6 Exception entry behavior

On pre-emption of an instruction stream, context state is saved by the hardware onto a stack pointed to by one of the SP registers (see *The SP registers* on page B1-8). The stack that is used depends on the mode of the processor at the time of the exception.

The stacked context supports the ARM Architecture Procedure Calling Standard (AAPCS). The support allows the exception handler to be an AAPCS-compliant procedure.

A full-descending stack format is used, where the stack pointer is decremented immediately before storing ~~a 32-bit word~~ (when pushing context) onto the stack, and incremented after reading ~~a 32-bit word~~ (popping context) from the stack. Eight 32-bit words are saved in descending order, with respect to their address in memory, as listed:

xPSR, `ReturnAddress()`, LR (R14), R12, R3, R2, R1, and R0

The exception entry pseudocode is:

```
// ExceptionEntry()
// ================

// NOTE: PushStack() can abandon memory accesses if a fault occurs during the stacking
//         sequence.
//         Exception entry is modified according to the behavior of a derived exception,
//         see DerivedLateArrival() and associated text.
```

```
PushStack();
ExceptionTaken(ExceptionType);    // ExceptionType is encoded as its exception number
```

For global declarations see *Register related definitions for pseudocode* on page B1-12.
For a definition of ExceptionActive[*] and NestedActivation see *Reset behavior* on page B1-20.
For helper functons and procedures see *Miscellaneous helper procedures and functions* on page AppxG-22.

The PushStack() and ExceptionTaken() pseudo-functions are defined as follows:

```
// PushStack()
// ===========

PushStack()
    if CONTROL<1> == '1' AND CurrentMode == Mode_Thread then
        frameptralign = SP_process<2> AND CCR.STKALIGN;
        SP_process = (SP_process - 0x20) AND NOT(ZeroExtend(CCR.STKALIGN:'00',32));
        frameptr = SP_process;
    else
        frameptralign = SP_main<2> AND CCR.STKALIGN;
        SP_main = (SP_main - 0x20) AND NOT(ZeroExtend(CCR.STKALIGN:'00',32));
        frameptr = SP_main;
       /* only the stack locations, not the store order, are architected */
    MemA[frameptr,4]      = R[0];
    MemA[frameptr+0x4,4]  = R[1];
    MemA[frameptr+0x8,4]  = R[2];
    MemA[frameptr+0xC,4]  = R[3];
    MemA[frameptr+0x10,4] = R[12];
    MemA[frameptr+0x14,4] = LR;
    MemA[frameptr+0x18,4] = ReturnAddress();
    MemA[frameptr+0x1C,4] = (xPSR<31:10>:frameptralign:xPSR<8:0>);
                            // see ReturnAddress() in-line note for information on xPSR.IT bits
    if CurrentMode==Mode_Handler then
        LR = 0xFFFFFFF1;
    else
        if CONTROL<1> == '0' then
            LR = 0xFFFFFFF9;
        else
            LR = 0xFFFFFFFD;
    return;


// ExceptionTaken()
// ================

ExceptionTaken(bits(9) ExceptionNumber)

    bit tbit;
    bits(32) tmp;

    R[0..3] = bits(32) UNKNOWN;
    R[12] = bits(32) UNKNOWN;
    tmp = MemA[VectorTable+4*ExceptionNumber,4];
    PC = tmp AND 0xFFFFFFFE;
    tbit = tmp<0>;
    CurrentMode = Mode_Handler;
    APSR = bits(32) UNKNOWN;                  // Flags UNPREDICTABLE due to other activations
    IPSR<8:0> = ExceptionNumber                   // ExceptionNumber set in IPSR
    EPSR.T = tbit;                                // T-bit set from vector
    EPSR.IT<7:0> = 0x0;                           // IT/ICI bits cleared
    /* PRIMASK, FAULTMASK, BASEPRI unchanged on exception entry*/
    CONTROL<1> = '0';                             // current Stack is Main, CONTROL<0> is unchanged
    /* CONTROL<0> unchanged */
    ~~NestedActivation = NestedActivation + 1;~~
    ExceptionActive[ExceptionNumber]= '1';
    SCS_UpdateStatusRegs();                       // update SCS registers as appropriate
    ClearExclusiveLocal();
    SetEventRegister()                            // see WFE instruction for more details
    InstructionSynchronizationBarrier();
```

For more details on the registers with UNKNOWN values, see *Exceptions on exception entry* on page B1-33. For updates to system status registers, see section *System Control Space (SCS)* on page B3-6.

ReturnAddress() is the address to which execution will return after handling of the exception:

```
// ReturnAddress()
// ===============

Bits(32) ReturnAddress() returns the following values based on the exception cause
         // NOTE: ReturnAddress() is always halfword aligned - bit<0> is always zero
         //       xPSR.IT bits saved to the stack are consistent with ReturnAddress()

// Exception Type          Address returned
// ==============          ================

// NMI:                      Address of Next Instruction to be executed
// HardFault (precise):      Address of the Instruction causing fault
// HardFault (imprecise):    Address of Next Instruction to be executed
// MemManage:                Address of the Instruction causing fault
// BusFault (precise):       Address of the Instruction causing fault
// BusFault (imprecise):     Address of Next Instruction to be executed
// UsageFault:               Address of the Instruction causing fault
// SVC:                      Address of the Next Instruction after the SVC
// DebugMonitor (precise):   Address of the Instruction causing fault
// DebugMonitor (imprecise): Address of Next Instruction to be executed
// IRQ:                      Address of Next Instruction to be executed after an interrupt
```

Note

A fault which is escalated to the priority of a HardFault retains the ReturnAddress() behavior of the original fault. For a description of priority escalation see *Priority escalation* on page B1-19.

IRQ includes SysTick and PendSV

B1.5.7 Stack alignment on exception entry

ARMv7-M supports a configuration option to ensure that all exceptions are entered with 8-byte stack alignment. The stack pointers in ARMv7-M are guaranteed to be at least 4-byte aligned. As exceptions can occur on any instruction boundary, it is possible that the current stack pointer is not 8-byte aligned when an exception activates.

The AAPCS requires that the stack-pointer is 8-byte aligned on entry to a conforming function[1]. Since it is anticipated that exception handlers will be written as AAPCS conforming functions, the system must ensure natural alignment of the stack for all arguments passed. The 8-byte alignment requirement is guaranteed in hardware using a configuration feature.

The STKALIGN bit (see *Configuration and Control Register (CCR)* on page B3-16) is used to enable the 8-byte stack alignment feature. Whether the bit is programmable in software and its value on reset are IMPLEMENTATION DEFINED . The bit should be set in the system boot sequence prior to needing 8-byte alignment support.

Note

Software must ensure that any exception handler that can activate while CCR.STKALIGN == '0' does not require 8-byte alignment. An example is an NMI exception entered from reset, where the implementation resets to 4-byte alignment.

If the bit is cleared between the entry to and return from an exception, and if the stack was not 8-byte aligned on entry to the exception, system corruption can occur. Support of a 4-byte aligned stack (CCR.STKALIGN == '0') in ARMv7-M is deprecated.

Theory of operation

On an exception entry when STKALIGN == 1, the stack pointer (SP_main or SP_process) in use before the exception entry is forced to have 8-byte alignment by adjusting its alignment as part of the exception entry sequence. The xPSR that is saved as part of the exception entry sequence records the alignment of this stack pointer prior to the exception entry sequence. The alignment status is merged and stored to memory as bit [9] of the xPSR (a reserved bit within the xPSR) in the saved context information.

On an exception exit when STKALIGN == 1, the stack pointer returned to takes its alignment from the value recovered from bit [9] of the xPSR in the restored context from the exception exit sequence. This reverses the forced stack alignment performed on the exception entry.

See *Exception entry behavior* on page B1-21 and *Exception return behavior* on page B1-25 for pseudocode details of the effect of the STKALIGN feature on exception entry and exception return.

[1] The AAPCS requires conforming functions to preserve the natural alignment of primitive data of size 1, 2, 4, and 8 bytes. In return, conforming code is permitted to rely on that alignment. To support unqualified reliance the stack-pointer must in general be 8-byte aligned on entry to a conforming function. If a function is entered directly from an underlying execution environment, that environment must accept the stack alignment obligation in order to give an unqualified guarantee that conforming code can execute correctly in all circumstances.

Note

Stack pointer alignment on exception exit is architecturally defined as an OR function. If the exception exit sequence is started with a stack pointer which is only 4 byte aligned, then this change has no effect.

In the event that the exception exit causes a derived exception, the derived exception is entered with the same stack alignment as was in use before the exception exit sequence started.

A side-effect when STKALIGN is enabled is that the amount of stack used on exception entry becomes a function of the alignment of the stack at the time that the exception is entered. As a result, the average and worst case stack usage will increase. The worst case increase is 4 bytes per exception entry.

Maintaining the stack alignment information in an unused bit within the saved xPSR makes the feature transparent to context switch code within operating systems, provided that the reserved status of unused bits with the xPSR have been respected.

Compatibility

Some operating systems can avoid saving and restoring R14 when switching between different processes in Thread mode if it is known that the values held in an EXC_RETURN value are invariant between the different processes. This provides a small improvement in context switch time, but at the cost of future compatibility. The STKALIGN feature does not affect the EXC_RETURN value. ARM does not guarantee that this software optimization will be possible in future revisions of the ARMv7-M architecture, and recommends for future compatibility that the R14 value is always saved and restored on a context switch.

B1.5.8 Exception return behavior

Exception returns occur when one of the following instructions loads a value of `0xFXXXXXXX` into the PC while in Handler mode:

- `POP`/`LDM` which includes loading the PC.
- `LDR` with `PC` as a destination.
- `BX` with any register.

When used in this way, the value written to the PC is intercepted and is referred to as the EXC_RETURN value.

EXC_RETURN[~~28~~:4] are reserved with the special condition that all bits should be written as one or preserved. Values other than all 1s are UNPREDICTABLE. EXC_RETURN[3:0] provide return information as defined in Table B1-8.

Table B1-8 Exception return behavior

EXC_RETURN[3:0]	
0bXXX0	RESERVED
0b0001	Return to Handler Mode; Exception return gets state from the Main stack; On return execution uses the Main Stack.
0b0011	RESERVED
0b01X1	RESERVED
0b1001	Return to Thread Mode; Exception return gets state from the Main stack; On return execution uses the Main Stack.
0b1101	Return to Thread Mode; Exception return gets state from the Process stack; On return execution uses the Process Stack.
0b1X11	RESERVED

RESERVED entries in this table result in a chained exception to a UsageFault.

If an EXC_RETURN value is loaded into the PC when in Thread mode, or from the vector table, or by any other instruction, the value is treated as an address, not as a special value. This address range is defined to have eXecute Never (XN) permissions, and will result in a MemManage exception, an INVSTATE UsageFault[1] exception, or the exception will escalate to a HardFault.

Integrity checks on exception returns

The ARMv7-M profile provides a number of integrity checks on an exception return. These exist as a guard against errors in the system software. Incorrect exception return information could be inconsistent with the state of execution which must be held in processor hardware or other state stored by the exception mechanisms.

The hardware related integrity checks ensure that the tracking of exception activation within the interrupt controller (NVIC) and System Control Block (SCB) hardware is consistent with the exception returns.

[1] It is IMPLEMENTATION DEFINED whether a MemManage or UsageFault exception occurs when an EXC_RETURN value is treated as a branch address, and bit [0] of the value is clear.

Integrity checks are provided to check the following conditions on an exception return:

- The Exception Number being returned from (as held in the IPSR at the start of the return) must be listed in the SCB as being active.
- ~~If no exceptions other than the returning exception are active, the mode being returned to must be Thread mode. This checks for a mismatch of the number of exception returns.~~
- If at least one exception other than the returning exception is active, under normal circumstances the mode being returned to must be Handler mode. This checks for a mismatch of the number of exception returns. This check can be disabled using the NONBASETHRDENA control bit in the SCB.
- On return to Thread mode, the Exception Number restored into the IPSR must be 0.
- On return to Handler mode, the Exception Number restored into the IPSR must not be 0.
- The EXC_RETURN[3:0] must not be listed as reserved in Table B1-8 on page B1-26

An exception return error causes an INVPC UsageFault, with the illegal EXC_RETURN value in the link register (LR).

Exception return operation

For global declarations see *Register related definitions for pseudocode* on page B1-12.
For `ExceptionTaken()` see *Exception entry behavior* on page B1-21.
For a definition of ExceptionActive[*] ~~and NestedActivation~~ see *Reset behavior* on page B1-20.
For helper functons and procedures see *Miscellaneous helper procedures and functions* on page AppxG-22.

```
// ExceptionReturn()
// =================

ExceptionReturn(bits(28) EXC_RETURN)
    assert CurrentMode == Mode_Handler;
    if !IsOnes(EXC_RETURN<27:4>) then UNPREDICTABLE;

    integer ReturningExceptionNumber = UInt(IPSR<8:0>);

    if ExceptionActive[ReturningExceptionNumber] == '0' then
        DeActivate(ReturningExceptionNumber);
        UFSR.INVPC = '1';
        LR = 0xF0000000 + EXC_RETURN;
        ExceptionTaken(UsageFault);                    // returning from an inactive handler
        return;
    else
        case  EXC_RETURN<3:0> of
            when '0001'                                // return to Handler
                if NestedActivation == 1 then
                    DeActivate(ReturningExceptionNumber);
                    UFSR.INVPC = '1';
                    LR = 0xF0000000 + EXC_RETURN;
                    ExceptionTaken(UsageFault);        // return to Handler exception mismatch
                    return;
                else
                    frameptr = SP_main;
                    CurrentMode = Mode_Handler;
                    CONTROL<1> = '0';
            when '1001'                                // returning to Thread using Main stack
                if NestedActivation != 1 && CCR.NONBASETHRDENA == '0' then
                    DeActivate(ReturningExceptionNumber);
                    UFSR.INVPC = '1';
                    LR = 0xF0000000 + EXC_RETURN;
                    ExceptionTaken(UsageFault);        // return to Thread exception mismatch
                    return;
                else
                    frameptr = SP_main;
                    CurrentMode = Mode_Thread;
                    CONTROL<1> = '0';
            when '1101'                                // returning to Thread using Process stack
                if NestedActivation != 1 && CCR.NONBASETHRDENA == '0' then
                    DeActivate(ReturningExceptionNumber);
                    UFSR.INVPC = '1';
                    LR = 0xF0000000 + EXC_RETURN;
                    ExceptionTaken(UsageFault);        // return to Thread exception mismatch
                    return;
                else
                    frameptr = SP_process;
                    CurrentMode = Mode_Thread;
                    CONTROL<1> = '1';
            otherwise
                DeActivate(ReturningExceptionNumber);
                UFSR.INVPC = '1';
                LR = 0xF0000000 + EXC_RETURN;
                ExceptionTaken(UsageFault);            // illegal EXC_RETURN
                return;

        DeActivate(ReturningExceptionNumber);
        PopStack(frameptr);

        if CurrentMode==Mode_Handler AND IPSR<8:0> =='000000000' then
            UFSR.INVPC = '1';
            PushStack();                              // to negate PopStack()
            LR = 0xF0000000 + EXC_RETURN;
            ExceptionTaken(UsageFault);               // return IPSR is inconsistent
            return;
        if CurrentMode==Mode_Thread AND IPSR<8:0> !='000000000' then
            UFSR.INVPC = '1';
            PushStack();                              // to negate PopStack()
            LR = 0xF0000000 + EXC_RETURN;
            ExceptionTaken(UsageFault);               // return IPSR is inconsistent
            return;

        ClearExclusiveLocal();
        SetEventRegister()                            // see WFE instruction for more details
        InstructionSynchronizationBarrier();
```

```
        if CurrentMode==Mode_Thread AND NestedActivation == 0 AND SCR.SLEEPONEXIT == '1' then
            PushStack();                             // to negate PopStack()
            SleepOnExit();                           // IMPLEMENTATION DEFINED
```

The DeActivate() and PopStack() pseudo-functions are defined as follows:

```
// DeActivate()
// ============

DeActivate(integer ReturningExceptionNumber)
    ExceptionActive[ReturningExceptionNumber] = '0';
    /* PRIMASK and BASEPRI unchanged on exception exit */
    if IPSR<8:0> != '000000010' then
        FAULTMASK<0> = '0';                  // clear FAULTMASK on any return except NMI
    ~~NestedActivation = NestedActivation - 1;~~
    return;


// PopStack()
// ==========

PopStack(bits(32) frameptr)   /* only stack locations, not the load order, are architected */
    R[0]  = MemA[frameptr,4];
    R[1]  = MemA[frameptr+0x4,4];
    R[2]  = MemA[frameptr+0x8,4];
    R[3]  = MemA[frameptr+0xC,4];
    R[12] = MemA[frameptr+0x10,4];
    LR    = MemA[frameptr+0x14,4];
    PC    = MemA[frameptr+0x18,4];         // UNPREDICTABLE if the new PC not halfword aligned
    psr   = MemA[frameptr+0x1C,4];
    case  EXC_RETURN<3:0> of
        when '0001'                          // returning to Handler
            SP_main = (SP_main + 0x20) OR ZeroExtend((psr<9> AND CCR.STKALIGN):'00',32);
        when '1001'                          // returning to Thread using Main stack
            SP_main = (SP_main + 0x20) OR ZeroExtend((psr<9> AND CCR.STKALIGN):'00',32);
        when '1101'                          // returning to Thread using Process stack
            SP_process = (SP_process + 0x20) OR ZeroExtend((psr<9> AND CCR.STKALIGN):'00',32);
    APSR<31:27> = psr<31:27>;                // valid APSR bits loaded from memory
    IPSR<8:0> = psr<8:0>;                    // valid IPSR bits loaded from memory
    EPSR<26:24,15:10> = psr<26:24,15:10>;    // valid EPSR bits loaded from memory
    return;
```

B1.5.9 Exceptions in single-word load operations

To support instruction replay, single-word load instructions must not update the destination register when a fault occurs during execution. By example, this allows replay of the following instruction:

```
LDR R0, [R2, R0];
```

B1.5.10 Exceptions in LDM and STM operations

In order to allow implementations to have the best possible interrupt response, an interrupt can be taken during an LDM or STM and continued after the return from the interrupt. The continuation state of the LDM or STM is held in the ICI bits in the EPSR (see *The special-purpose program status registers (xPSR)* on page B1-8). It is IMPLEMENTATION DEFINED when interrupts are recognized, so the use of the ICI bits is IMPLEMENTATION DEFINED.

The ARMv7-M architecture supports continuation of, or restarting from the beginning, an abandoned LDM or STM instruction as outlined below. Where an LDM or STM is abandoned and restarted (ICI bits are not supported), the instructions should not be used with volatile memory. To support instruction replay, the LDM, STM, PUSH and POP instructions are required to restore/maintain the base register when a fault occurs (no base register writeback update) during execution.

Note

LDM and STM instructions where the ICI bits are not supported are incompatible with the single-copy atomicity rules for Device and Strongly Ordered memory described in *Memory access restrictions* on page A3-26.

The ICI bits encode the number of the first register in the register list that must be loaded or stored on return to an LDM or STM instruction. When the LDM or STM is returned to, all registers of an equal or higher number in the instruction register list to the value held in the ICI bits are loaded/stored.

The continuation register is encoded as:

Bits [26:25] = '00';

Bits [15:12] = RegisterNumber;

Bits [11:10] = '00';

When Bits [26:25,15:10] are all zero, this does not represent a continuation state. It encodes normal operation with neither IT nor interrupt continuation active. An abandoned instruction restarts.

If the register number held in the ICI bits is non-zero and is not a register in the register list of the LDM or STM the result is UNPREDICTABLE.

If the register number held in the ICI bits is non-zero and is the first register in the register list of the LDM or STM the result is UNPREDICTABLE.

If the ICI bit field is non-zero, and the instruction executed on an exception return is not an LDM, STM or within an IT block, a UsageFault (INVSTATE, see *Fault behavior* on page B1-39) is generated.

If a fault (BusFault or MemManage) occurs on any LDM or STM instruction (including PUSH and POP), the instruction is abandoned, and will be restarted from the beginning on return from the exception. If the instruction is not within an IT block, the ICI bits are cleared to zero.

Note

The IT feature takes precedence over the ICI feature if an exception-continuable instruction is used within an IT construct. In this situation, the multi-cycle load or store instruction is treated as restartable, and should not be used with Device or Strongly Ordered memory.

The following sections describe restrictions that apply to taking an exception during an `LDM` or `STM`:

Load multiple and PC in load list

For the ARM architecture in general, the case of `LDM` with PC in the register list is defined to be unordered, allowing the registers to be loaded in a different order than the register mask implies. The usual use is to allow the PC to be loaded first.

For ARMv7-M, however, `LDM` operations with the PC in the register list can be interrupted during their operation, and the continuation state held in the ICI bits. On an exception return to an `LDM` instruction, the ICI bits indicate which register must be loaded next to continue properly; this can result in an `LDM` with the PC in the register list accessing the same location in memory twice.

If the PC was loaded early, the PC presented to the exception entry sequence must be restored such that the return address from the exception taken is to the `LDM` instruction address; it is then loaded again when the `LDM` is continued.

Load-store multiple, base register update and the ICI bits

The base register can be changed as a result of a load or store multiple under the following situations:

- Base register write-back: ~~see load/store instruction writeback details in Chapter A6 *Thumb Instruction Details* for more information~~.
- Base load: the base register is one of the registers in the register list of an `LDM`.

The value left in the base register in the case of an exception which occurs during the load or store multiple instruction is as follows:

Fault condition (BusFault or MemManage):

- Applies to all forms of `LDM` and `STM`, including `PUSH` and `POP`
  - — The base register is restored to the original value.
  - — If the instruction is not within an IT block, the ICI bits are cleared to zero.
  - — If the instruction is within an IT block, the ICI bits are not used to hold the continuation state as the IT bits indicate the position in the IT block.
  - — In all cases, a return from the fault handler will fully restart the instruction.

Base register write-back:

- Interrupt of an `LDM` or `STM` in an IT block:
  - — The base register contains the initial value, whether an IA or DB `LDM`/`STM` instruction.

  - The ICI bits are not used to hold the continuation state, as the IT bits indicate the position in the IT block.
- Interrupt of an `LDM` or `STM`, not in an IT block, using SP as the base register.
  - The SP that is presented to the exception entry sequence is lower than any element pushed (`STM`) or not yet popped (`LDM`).

    For instructions decrementing before (DB), the SP is set to the final (lowest) value.

    For instructions incrementing after (IA), the SP is set to the initial (lowest) value. In all cases, the ICI bits hold the continuation state.
- Interrupt of `LDM` or `STM` not in an IT block and not using SP as the base register:
  - The base register contains the final value, whether an IA or DB `LDM`/`STM` instruction.
  - The ICI bits hold the continuation state.

Base register load

- In all cases, the original base address is restored when the instruction is abandoned
- ~~Volatile locations should not be accessed with `LDM` operations that:~~
  - ~~execute inside an IT block~~
  - ~~load the base register~~
  - ~~load the PC~~
- ~~Interrupt of an `LDM` in an IT block:~~
  - ~~If the instruction is in an IT block, the ICI bits cannot be used to hold the continuation state, as the IT bits indicate the position of the instruction in the IT block. It is IMPLEMENTATION DEFINED whether the instruction executes to completion or restarts.~~
- ~~Interrupt of an `LDM` not in an IT block:~~
  - ~~If the interrupt activates before the base has been loaded, implementations can use the ICI bits to hold the continuation state.~~
  - ~~If the interrupt activates after the base register has been loaded, implementations must restore the base register to its original value. The ICI bits can be set to an IMPLEMENTATION DEFINED value that will load at least the base register and subsequent locations again on return.~~

As a base register load example, if `LDM` R2, {R0-R4}, not in an IT block, is interrupted:

- ~~after R0 or R1 has been loaded and continuation supported: the continuation bits indicate a restart on R1 or R2~~
- ~~after R2 has been loaded and continuation supported: the continuation bits indicate a restart on R1 or R2~~
- ~~after R3 has been loaded and continuation supported: the continuation bits indicate a restart on R1 or R2~~

- ~~in all cases where continuation is not supported and the instruction is abandoned prior to loading R4, the instruction will restart with the ICI bits cleared to zero.~~

B1.5.11 Exceptions on exception entry

During Exception Entry other exceptions can occur, either because of a fault on the operations involved in exception entry, or because of the arrival of an asynchronous exception, an interrupt, which is of higher priority than the exception entry sequence in progress.

Late arriving exceptions

The ARMv7-M profile does not specify the point at which the arrival of an asynchronous exception is recognized during an exception entry. However, in order to support implementations with very low interrupt latencies, the ARMv7-M profile provides some facilities to permit high priority interrupts arriving during an exception entry to activate during the exception entry, and for the entry sequence not to be repeated.

When an asynchronous interrupt activates during the exception entry sequence, the exception that caused the exception entry sequence is known as the original exception. The exception caused by the interrupt is known as the ~~secondary~~ exception.

It is permissible in this case for the exception entry sequence that was started by the original exception to be used by the ~~secondary~~ exception. The original exception is taken after the ~~secondary~~ exception has returned. This is referred to as late-arrival pre-emption.

For a late arrival pre-emption, the ~~secondary~~ exception ~~(interrupt, fault or supervisor call)~~ is entered and the original exception is left in the Pending state.

It is IMPLEMENTATION DEFINED what conditions, if any, lead to late arrival pre-emption. Late arrival pre-emption can only occur when the ~~secondary~~ exception is of higher priority than the original exception. ~~Where late arrival exceptions are supported:~~

```
~~// LateArrival()~~
~~// ============~~

~~LateArrival()~~

    ~~// xEpriority: the lower the value, the higher the priority~~

    ~~integer OEpriority;  // original exception group priority~~
    ~~integer SEpriority;  // secondary exception group priority~~
    ~~integer OEnumber;    // ExceptionNumber for OE~~
    ~~integer SEnumber;    // ExceptionNumber for SE~~


    ~~if (SEpriority < OEpriority) then~~
        ~~ExceptionTaken(SEnumber);        // secondary exception taken~~
    ~~else~~
        ~~ExceptionTaken(OEnumber);        // original exception taken~~
```

For `ExceptionTaken()` ~~and~~ ~~`PushStack()`~~ see *Exception entry behavior* on page B1-21.

Derived exceptions

Where an exception entry sequence itself causes a fault, the exception that caused the exception entry sequence is known as the original exception. The fault that is caused by the exception entry sequence is known as the derived exception. The code stream that was running at the time of the original exception is known as the pre-empted code whose execution priority is the pre-empted priority.

The following derived exceptions can occur during exception entry:

- a MemManage fault on the writes to the stack memory as part of the exception entry (this is described as a MSTKERR class of MemManage fault)
- a BusFault on the stack on the writes to the stack memory as part of the exception entry (this is described as a STKERR class of BusFault)
- a watchpoint can ~~give rise to~~ a DebugMonitor fault on exception entry
- a BusFault on reading the vector (this is always treated as a HardFault).

If the pre-empted group priority is higher than or equal to the group priority of the derived exception then:

- if the Derived Exception was DebugMonitorFault, the exception is ignored
- if the Derived Exception was not DebugMonitorFault, the derived exception is escalated to HardFault.

Note

Note: the priority of the original exception is not involved.

Derived exceptions are treated similarly to late arriving exceptions and it is permissible for implementations to use late arrival pre-emption. Late arrival pre-emption can only occur when the derived exception (after escalation if appropriate) is of higher priority than the original exception, but it is IMPLEMENTATION DEFINED exactly what conditions, if any, lead to late arrival pre-emption.

If the late-arrival pre-emption approach is not used, the derived exception is set to Pending, and the exception will be taken in accordance with the prioritization rules for Pending exceptions.

If late-arrival pre-emption is used, the derived exception ~~(interrupt, fault or Supervisor call)~~ is entered and the original exception is left in the Pending state.

The ~~behavior can be summarized as follows:~~

For `ExceptionTaken()` and `PushStack()` see *Exception entry behavior* on page B1-21.

```
~~// DerivedLateArrival()~~
~~// ====================~~

~~DerivedLateArrival()~~

    ~~// xEpriority: the lower the value, the higher the priority~~
    ~~// PE: the pre-empted exception - before exception entry~~
    ~~// OE: the original exception - exception entry~~
    ~~// DE: the derived exception - fault on exception entry~~
```

```
~~integer PEpriority;  // pre-empted exception group priority~~
~~integer OEpriority;  // group priority of the original exception~~
~~integer DEpriority;  // derived exception group priority~~

~~integer PEnumber;    // ExceptionNumber for PE~~
~~integer OEnumber;    // ExceptionNumber for OE~~
~~integer DEnumber;    // ExceptionNumber for DE~~

~~boolean DEisDbgMonFault;         // DE is a DebugMonitorFault~~

~~if DEpriority < PEpriority && DEisDbgMonFault then~~
    ~~ExceptionTaken(OEnumber);     // ignore the DebugMonitor fault~~
~~if DEpriority < PEpriority && !DEisDbgMonFault then~~
    ~~DEpriority = -1;              // escalate DE to HardFault~~
                                  ~~// (incl. BKPT with DebugMonitor disabled)~~
    ~~SetPending(OEnumber);         // OE to Pending state~~
    ~~ExceptionTaken(HardFault);~~
~~else~~
    ~~if DEpriority < OEpriority then~~
        ~~SetPending(OEnumber);     // OE to Pending state~~
        ~~ExceptionTaken(DEnumber); // start execution of the DE~~
                                  ~~// tail-chaining IMPLEMENTATION DEFINED~~
    ~~else~~
        ~~SetPending(DEnumber);     // DE to Pending state~~
        ~~ExceptionTaken(OEnumber); // start execution of the OE~~
```

Pending state information is maintained in *Interrupt Control State Register (ICSR)* on page B3-12 and *System Handler Control and State Register (SHCSR)* on page B3-18.

Note

It is IMPLEMENTATION DEFINED whether late-arrival exceptions are supported and can affect derived exceptions. Where late-arrival exceptions are supported, DE maps to OE and the late-arrival exception maps to SE in the late-arrival pseudocode (see *Late arriving exceptions* on page B1-33).

B1.5.12 Tail-chaining and exceptions on exception return

During exception return, other exceptions can affect behavior, either because of a fault on the operations involved in exception return, or because of an asynchronous exception that is of higher priority than the priority level being returned to during the exception return. The asynchronous exception can be already Pending or arrive during the exception return.

The target of the exception return is described by the Exception Return Link. The target priority is the highest priority active exception, excluding the exception being returned from, or the boosted priority set by the special-purpose mask registers, whichever is higher.

Derived exceptions

Where an exception return sequence causes a fault exception, the exception caused by the exception return sequence is known as the derived exception.

The following derived exceptions can occur during exception return:

- a MemManage fault on the reads to the stack memory as part of the exception return (this is described as a MUNSTKERR class of MemManage fault)
- a BusFault on the stack on the reads to the stack memory as part of the exception return (this is described as an UNSTKERR class of BusFault)
- a watchpoint can give rise to a DebugMonitorFault on exception return
- ~~an integrity check on the exception return causing a UsageFault.~~

If the target group priority is higher than or equal to the group priority of the derived exception, then:

- if the derived exception is a DebugMonitorFault, it is ignored
- if the derived exception is not a DebugMonitorFault, the derived exception is escalated to HardFault.

In the event of a derived exception, the derived exception is entered using tail-chaining.

Tail-chaining

Tail-chaining is the optimization of an exception return and an exception entry so that the loads and stores of the key context state can be eliminated.

Tail-chaining is used for two reasons:

- For handling derived exceptions
- As an optimization that implementations are permitted to use to improve interrupt response when there is a Pending exception which has a higher group priority than the target group priority. In this case, the architected behavior is that the Pending exception will be taken immediately on exception return, and tail-chaining permits the optimization of the return and entry sequences.

In the tail-chaining optimization, the exception return and exception entry sequences are combined to form the following sequence, where the ReturningExceptionNumber is the number of the exception being returned from, and the ExceptionNumber is the number of the exception being tail-chained to. EXC_RETURN is the EXC_RETURN value that caused the original exception return to start.

For `ExceptionTaken()` see *Exception entry behavior* on page B1-21.
For `DeActivate()` see *Exception return behavior* on page B1-25.

```
// TailChain()
// ===========

TailChain(bits(28) EXC_RETURN)
    assert CurrentMode == Mode_Handler;
    if !IsOnes(EXC_RETURN<27:4>) then UNPREDICTABLE;

    integer ReturningExceptionNumber = UInt(IPSR<8:0>);
    LR = 0xF0000000 + EXC_RETURN;
    DeActivate(ReturningExceptionNumber);
    ExceptionTaken(ExceptionNumber);
```

```
/* NestedActivation is effectively unchanged by a tail-chain */
```

Use of tail-chaining as an optimization for pending exceptions

The use of tail-chaining as an optimization for performing exception returns when there are Pending exceptions has a behavior which is different from simply performing the exception return, followed by the Pending exception entry. The difference in behavior is that many of the derived exceptions that could occur as a result of the exception return and the exception entry might not occur. Instead, these derived exceptions will occur when the pending exception is returned from.

Late arrival pre-emption and tail-chaining during exception returns

The ARMv7-M profile does not specify the point at which arrival of asynchronous exceptions are recognized during an exception. The ARMv7-M profile permits exceptions of a higher priority than the priority of the exception to be tail-chained to, to be entered in place of that exception being tail-chained to, using late-arrival pre-emption. It is IMPLEMENTATION DEFINED what conditions, if any, lead to late arrival pre-emption.

Late-arrival pre-emption can occur during a tail-chaining execution sequence due to a derived exception on an exception return. The derived exception is marked as Pending when a late-arrival pre-emption of the derived exception occurs.

B1.5.13 Exception status and control

The System Control Block within the System Control Space (*The System Control Block (SCB)* on page B3-10) provides the register support required to manage the exception model. The registers break down into the following categories:

- General system configuration, status and control
  - Vector Table Offset Register – see *The vector table* on page B1-16
  - Interrupt Control State Register – see *Interrupt Control State Register (ICSR)* on page B3-12
  - Application Interrupt and Reset Control Register – see *Application Interrupt and Reset Control Register (AIRCR)* on page B3-14
  - System Control Register– see *Power management* on page B1-48
  - Configuration Control Register – see *Configuration and Control Register (CCR)* on page B3-16
  - System Handler Priority Registers – see *System Handler Priority Register 1 (SHPR1)* on page B3-17, *System Handler Priority Register 2 (SHPR2)* on page B3-17, and *System Handler Priority Register 3 (SHPR3)* on page B3-17
  - System Handler Control and State Register – see *System Handler Control and State Register (SHCSR)* on page B3-18
  - Fault handling support – see *Fault behavior* on page B1-39 and *Fault status and address information* on page B1-42 for details

  - Software Trigger Interrupt Register – see *Software Trigger Interrupt Register (STIR)* on page B3-23

- SysTick support – see *System timer - SysTick* on page B3-24
- NVIC support – see *Nested Vectored Interrupt Controller (NVIC)* on page B3-28

The Interrupt Control State Register (*Interrupt Control State Register (ICSR)* on page B3-12) provides the ability to set the Pending state in software for NMI, SysTick and PendSV. It also provides the ability to clear the Pending state in software for SysTick and PendSV, and provides status information of pending and active exceptions.

The Application Interrupt and Reset Control Register (*Application Interrupt and Reset Control Register (AIRCR)* on page B3-14) provides priority grouping control for the exception model, endian status for data accesses, and reset controls (see *Reset management* on page B1-47 for more details). Endianness can only be configured at reset and not under software control. See *Control of the Endian Mapping in ARMv7-M* on page A3-6 for more details. The register includes a vector key field that must be written with the key value `0x05FA` for a write transaction to be accepted.

The Configuration Control Register (*Configuration and Control Register (CCR)* on page B3-16) provides configuration control for:

- Enabling divide by zero faults, alignment faults and some operation controls
- Disabling BusFaults at priority -1 and above

The System Handler Priority Registers (*System Handler Priority Register 1 (SHPR1)* on page B3-17, *System Handler Priority Register 2 (SHPR2)* on page B3-17 and *System Handler Priority Register 3 (SHPR3)* on page B3-17) provide mechanisms to program the priority of BusFault, MemManage, UsageFault, Debug Monitor Fault, SVCall, SysTick and PendSV.

The System Handler Control and State Register (*System Handler Control and State Register (SHCSR)* on page B3-18) provides access to the Pending and Active status of faults and supervisor calls plus the active status of the SysTick and PendSV interrupts. This register provides the ability to read and write the state bits as part of a context switch. The register also provides the ability to enable the UsageFault, BusFault and MemManage exception handlers. When the fault handlers are disabled, the faults are escalated (see *Priority escalation* on page B1-19).

Note

- There are no explicit active state bits for the fixed priority exceptions (reset, NMI or HardFault).
- The debug monitor is enabled in a debug control register (see *Debug Exception and Monitor Control Register (DEMCR)* on page C1-24).
- SysTick is enabled in a SysTick control register (see *SysTick Control and Status Register (SYST_CSR)* on page B3-26).
- The active and pending state bits are provided to support the save and restore of information on a context switch. In particular, for explicit (software) writes to the System Handler Control and State Register:
  - setting an active bit does not cause an exception entry

- — clearing an active bit does not cause an exception return
- — setting a pending bit for an exception in this register when the execution group priority is lower than the group priority of the exception associated with the pending bit is UNPREDICTABLE.

The Software Trigger Interrupt Register (*Software Trigger Interrupt Register (STIR)* on page B3-23) is a write only register, which provides a general method for setting a pending register by its exception number. Only external interrupts can be pended by this method. Attempts to write an exception number in the range 0-15 are ignored. Attempts to trigger an interrupt number not supported by a core are also ignored.

The NVIC registers (*NVIC register support in the SCS* on page B3-30) provide the following functions for external interrupts:

- enabling and disabling
- setting and clearing the Pending state
- reading the Active state
- programming the priority.

Note

Interrupts can become Pending when the associated interrupt is disabled. Enabling an interrupt allows a Pending interrupt to activate.

B1.5.14 Fault behavior

In accordance with the ARMv7-M exception priority scheme, precise fault exception handlers execute in one of the following ways:

- taking the specified exception handler
- taking a HardFault exception
- in the case of a fault arising while executing at priority "-1" or above, as described in *Unrecoverable exception cases* on page B1-44.

Fault handling by a dedicated handler or HardFault handler can be summarized as follows:

```
// Fault handling
// ==============

// FaultType is a subset of ExceptionNumber and can be one of the following values:

bits(9) HardFault = 3;
bits(9) MemManage = 4;
bits(9) BusFault = 5;
bits(9) UsageFault = 6;

ExceptionTaken(FaultType);
```

In all fault handling cases, the corresponding fault status register bit is set, and the fault handler will return according to the rules defined in ReturnAddress(), see *Exception entry behavior* on page B1-21 for more details.

Table B1-9 lists all faults. The information provided includes the cause, exception taken, the name of the associated fault status bit, and which debug vector catch bit (if any) is used to catch the associated fault.

Table B1-9 List of supported faults

Fault Cause	Fault exception	Bit Name in the HFSR or CFSR [Note 1]	Notes	Debug vector catch bit
Vector Read error	HardFault	VECTTBL	Bus error returned when reading the vector table entry. Exception vector reads use the default address map – see *Protected Memory System Architecture (PMSAv7)* on page B3-35	INTERR
Fault escalation	HardFault	FORCED	Fault or supervisor call occurred, and the handler group priority is lower or equal to the execution group priority. The exception escalates to a HardFault. Fault address and status registers (as appropriate) are updated.	HARDERR
Breakpoint (BKPT) escalated	HardFault	DEBUGEVT	Occurs when halting debug and the DebugMonitor are disabled, and a BKPT associated exception is escalated.	HARDERR
BusFault on exception entry stack memory operations	BusFault	STKERR	Failure when saving context via hardware – bus error returned. The BusFault Address Register is not written by this fault.	INTERR
MemManage fault on exception entry stack memory operations	MemManage	MSTKERR	Failure when saving context via hardware – MPU access violation. The MemManage Address Register is not written by this fault.	INTERR
BusFault on exception return stack memory operations	BusFault	UNSTKERR	Failure when restoring context via hardware – bus error returned. The Bus Fault Address Register is not written by this fault.	INTERR

Table B1-9 List of supported faults (continued)

Fault Cause	Fault exception	Bit Name in the HFSR or CFSR [Note 1]	Notes	Debug vector catch bit
MemManage fault on exception return stack memory operations	MemManage	MUNSTKERR	Failure when restoring context via hardware – MPU access violation. The MemManage Address Register is not written by this fault.	INTERR
MemManage fault on data access	MemManage	DACCVIOL	Violation/fault on MPU due to an explicit memory access. The MemManage Address Register is written with the data address of the load/store.	MMERR
MemManage fault on instruction access	MemManage	IACCVIOL	Violation/fault on MPU due to instruction fetch. Includes fetches from XN memory when no MPU. Faults only if the processor attempts to execute the instruction.The MemManage Address Register is not written by this fault.	MMERR
Bus error on instruction access (precise)	BusFault	IBUSERR	Bus error returned on an instruction fetch. Faults only if the processor attempts to execute the instruction. The Bus Fault Address Register is not written by this fault.	BUSERR
Precise bus error on data access	BusFault	PRECISERR	Precise bus error due to an explicit memory access. The Bus Fault Address Register is written with the data address of the load/store.	BUSERR
Imprecise bus error on data bus	BusFault	IMPRECISERR	Imprecise bus error due to an explicit memory access. The Bus Fault Address Register is not written by this fault.	BUSERR
No Coprocessor	UsageFault	NOCP	Does not exist, or access denied (see *Coprocessor Access Control Register (CPACR)* on page B3-22)	NOCPERR
~~Undefined~~ Instruction	UsageFault	UNDEFINSTR	Unknown instruction (including those associated with an enabled Coprocessor)	STATERR

Table B1-9 List of supported faults (continued)

Fault Cause	Fault exception	Bit Name in the HFSR or CFSR [Note 1]	Notes	Debug vector catch bit
Attempt to execute an instruction when EPSR.T==0	UsageFault	INVSTATE	Attempt to execute in an invalid EPSR state (e.g. after a BX type instruction has changed state). This includes state change after entry to or return from exception, as well as from inter-working instructions.	STATERR
Exception return integrity check failures	UsageFault	INVPC	Any failures of the integrity checks for exception return listed in *Integrity checks on exception returns* on page B1-26	STATERR
Illegal unaligned load or store	UsageFault	UNALIGNED	This will occur when any load-store multiple instruction attempts to access a non-word aligned location. It will also occur for any load-store if it is not naturally aligned and the UNALIGN_TRP bit is set. [Note 2]	CHKERR
Divide By 0	UsageFault	DIVBYZERO	This will occur when SDIV or UDIV is executed with a divisor of 0, and the DIV_0_TRP bit is set. [Note 2]	CHKERR
For debug related faults – see Chapter C1 *ARMv7-M Debug*				
Note 1: CFSR = Configurable Fault Status register, HFSR = HardFault Status register (see below) Note 2: UNALIGN_TRP and DIV_0_TRP are control bits in the Configuration and Control register				

Fault status and address information

The System Control Space includes the following fault status and fault address registers.

- Configurable Fault Status registers for UsageFault, BusFault and MemManage
- A HardFault Status register – see *HardFault Status register (HFSR)* on page B3-21 for more details
- A Debug Fault Status register – see *Debug Fault Status Register (DFSR)* on page B3-21 and Chapter Chapter C1 *ARMv7-M Debug* for more details

- BusFault and MemManage Fault Address registers (*BusFault Address Register (BFAR)* on page B3-22 and *MemManage Address Register (MMFAR)* on page B3-22 respectively). It is IMPLEMENTATION DEFINED whether the fault address registers are unique registers, or are a shared resource accessible from two locations in the System Control Space.

The HardFault Status register supports three fault handling status flags, indicating the reason for taking a HardFault exception. All status flags are write-1-to-clear.

The 32-bit Configurable Fault Status register (*Configurable Fault Status Registers (UserFault, BusFault, and MemManage)* on page B3-18) is a concatenation of fault status for the UsageFault, BusFault, and MemManage status registers. UsageFault, BusFault and MemManage exceptions are known as configurable faults, as they all support dynamic priority setting. Fault Status register bits are additive – each new fault sets a bit. All status flags are write-1-to-clear.

The BusFault and MemManage status registers include a valid bit which is set when the associated fault address register is updated. The MemManage Address register is updated with the faulting address for data access violations only. The BusFault Address register is updated with the faulting address in the case of precise data errors only. The address of the faulting instruction for UsageFault, MemManage and Precise BusFaults can be determined from the stacked ReturnAddress() as defined in *Exception entry behavior* on page B1-21.

Note

- The escalation of BusFault or MemManage to HardFault can cause the associated fault address register to be overwritten by a derived exception (see *Exceptions on exception entry* on page B1-33 and *Tail-chaining and exceptions on exception return* on page B1-35). Fault handlers must ensure that the valid bit is checked and cleared by the HardFault handler when a derived exception causes this corruption.
- There are cases where the fault address register will not be valid. Handlers should check address validity by ensuring its associated VALID bit is set. An invalid address can occur due to pre-emption of a fault.

The Configuration and Control register (*Configuration and Control Register (CCR)* on page B3-16) includes control bits for three fault related features:

- a control bit (BFHFNMIGN) to inhibit data access bus faults when executing at priority -1 or -2.

  An example use is to allow autoconfiguration of bridges and devices where probing of disabled or non-existent elements can cause bus faults. Software must ensure the executing exception handler's code and data space are valid for correct operation.
- a control bit (DIV_0_TRP) to enable divide-by-0 traps – see Table B1-9 on page B1-40.
- a control bit (UNALIGN_TRP) to enable unaligned access traps on words and halfwords – see Table B1-9 on page B1-40.

B1.5.15 Unrecoverable exception cases

The ARMv7-M profile generally assumes that when the processor is running at priority -1 or above, any faults or supervisor calls that occur are fatal and are entirely unexpected.

The standard exception entry mechanism does not apply where a fault or supervisor call occurs at a priority of -1 or above. ARMv7-M handles most of these cases using a mechanism called lock-up, otherwise the condition becomes Pending or is ignored. Lock-up suspends normal instruction execution and enters lock-up state. When in lock-up state, the behavior is:

- The processor repeatedly fetches the same instruction, from a fixed address, determined by the exact nature of the fault, as described in Table B1-10 on page B1-45.
- The instruction fetched is executed repeatedly if it is a valid instruction. If the lock-up is caused by a precise memory error on a load or store which has base write-back, the base register is restored by the fault.
- If the IT bits are non-zero at the time that lock-up occurs, the IT bits are not advanced.
- The S_LOCKUP bit in the Debug Halting Control and Status register is set.
- The Fault Status Register bits consistent with the fault causing the lock-up will be set.

It is strongly recommended that implementations provide an external signal which indicates that the lock-up state has been entered to allow external mechanisms to react.

The lock-up state can be exited in one of 4 ways:

- If in a HardFault handler an NMI exception occurs, the NMI will be activated as normal. The NMI return link will be the address used for the lock-up state.
- A System reset occurs. This will exit lock-up state and reset the system as normal.
- A halt command from a halt mode debug agent is issued. The core will enter Debug state with the PC set to the same value as is used for return context. See Table B1-10 on page B1-45 for details.
- A memory error that can be resolved by the system through specific action(s) or over time (by example, a resource that requires time to configure).

In most cases, once in lock-up state, the processor will continue in this manner until reset (such as from a watchdog). However, the reason for the lock-up state can be examined and corrected without a reset by one of the following mechanisms:

- The system is stopped by a debugger (Halt issued, Debug state entered) with a corresponding fix to the xPSR, instruction, ~~FaultMask~~ and/or PC.
- If the problem is due to fetch errors (BusFault on read), an external master can correct the problem or the problem self-corrects if it is transitory.
- If the problem is due to an ~~undefined~~ instruction, an external master can modify the memory contents.

- If NMI pre-empts lock-up state in a HardFault exception, the NMI handler can fix the problem before returning (e.g. changing the return PC, changing the value in the FAULTMASK register and/or fixing the state bits in the saved xPSR).

In these cases, the processor will leave the lock-up state and continue executing from the lock-up or modified (PC) address as stated above.

Note

While the behavior described is architecturally defined, none of the above are suggested as application approaches, as the lockup conditions are expected to be terminal.

Table B1-10 outlines the behavior of all faults or supervisor calls that can occur during HardFault or NMI handler execution. Where the system locks up at a priority of -1, it is IMPLEMENTATION DEFINED whether the ~~EPSR~~ indicates Hardfault and/or if the FAULTMASK bit is set.

Table B1-10 Behavior of faults which occur during NMI or HardFault execution

Fault cause	Occurrence	Behavior	Lock-up Address	Notes
VECTABLE read error at reset	Cannot read vector table for SP or PC at reset	Lock-up at priority -1	`0xFFFFFFFF`	
VECTABLE read error on NMI entry	Cannot read NMI vector	Lock-up at priority -2	`0xFFFFFFFF`	
VECTABLE read error on HardFault entry	Cannot read HardFault vector	Lock-up at priority -1	`0xFFFFFFFF`	
BusFault – Instruction	Priority -1 or -2	Lock-up at priority of occurrence	Faulting instruction	Can auto-correct if bus fault is transitory.
BusFault – Imprecise Data	Priority -1 or -2	Imprecise Bus Fault is set to the Pending state		Does not lock up
BusFault – Precise Data	Priority -1 or -2	Configurable: Lock-up at priority of occurrence or ignored using BFHFNMIGN	Faulting instruction	BFHFNMIGN[a]== 0: Lock-up state is entered BFHFNMIGN == 1: BFSR[b] bits will be set but otherwise the BusFault will be ignored These errors can auto-correct.

Table B1-10 Behavior of faults which occur during NMI or HardFault execution (continued)

Fault cause	Occurrence	Behavior	Lock-up Address	Notes
BusFault – STKERR on NMI entry	Priority before NMI was -1.	Lock-up at priority -1 or -2 IMPLEMENTATION DEFINED	`0xFFFFFFFF`	
BusFault – UNSTKERR	Un-stacking from an NMI to target priority of -1	Lock-up at priority -1	`0xFFFFFFFF`	Continually fetching `0xFFFFFFFF` at priority = -1 (Hard Fault).
MemManage – Instruction	Priority -1 or -2	Lock-up at priority of occurrence	Faulting instruction	HFNMIENA[c] controls whether this can occur from the MPU. HFNMIENA == 0: lock-up can occur from the default memory map's XN partitions. HFNMIENA == 1: lock-up can occur when a fetch causes an MPU Access violation, XN region violation, or missing region fault.
MemManage – Data	Priority -1 or -2	Lock-up at priority of occurrence	Faulting instruction	HFNMIENA == 0: lock-up cannot occur HFNMIENA == 1: lock-up can occur from MPU access or privilege violation
MemManage – MSTKERR on NMI entry	Priority before NMI was -1.	Lock-up at priority -1 or -2 IMPLEMENTATION DEFINED	`0xFFFFFFFF`	HFNMIENA == 0: lock-up cannot occur HFNMIENA == 1: lock-up can occur from MPU access or privilege violation
MemManage – MUNSTKERR on NMI return	Un-stacking from an NMI to target priority of -1	Lock-up at priority -1	`0xFFFFFFFF`	HFNMIENA == 0: lock-up cannot occur HFNMIENA == 1: lock-up can occur from MPU access or privilege violation

Table B1-10 Behavior of faults which occur during NMI or HardFault execution (continued)

Fault cause	Occurrence	Behavior	Lock-up Address	Notes
SVC	Priority -1 or -2	Lock-up at priority of occurrence	Faulting instruction	At priority -1 or -2, SVC is treated as an UNDEFINED instruction.
Usage Fault – all, except INVPC	At Priority -1 or -2	Lock-up at priority of occurrence	Faulting instruction	
Usage Fault – INVPC	On un-stacking from NMI to target priority of -1.	Lock-up at priority -2	`0xFFFFFFFF`	
Usage Fault – INVPC	On return from Priority -1	Tailchains to HardFault	N/A	The fault on the exception return causes re-entry to the HardFault exception handler. This is not a lock-up condition.
Breakpoint[d]	At Priority -1 or -2	Lock-up at priority of occurrence	Breakpoint instruction	

a. See *Configuration and Control Register (CCR)* on page B3-16

b. See *Configurable Fault Status Registers (UserFault, BusFault, and MemManage)* on page B3-18

c. See *MPU Control Register (MPU_CTRL)* on page B3-40

d. `BKPT` instruction or FPB (see *Flash Patch and Breakpoint (FPB) support* on page C1-61) generated.

B1.5.16 Reset management

The Application Interrupt and Reset Control register (*Application Interrupt and Reset Control Register (AIRCR)* on page B3-14) provides two mechanisms for a system reset:

- The control bit SYSRESETREQ requests a reset by an external system resource. The system components which are reset by this request are IMPLEMENTATION DEFINED. SYSRESETREQ is required to cause a Local Reset.
- The control bit VECTRESET (a debug feature, see *Reset and debug* on page B1-48 below) causes a Local Reset. It is IMPLEMENTATION DEFINED whether other parts of the system are reset as a result of this control.

Note

SYSRESETREQ and VECTRESET should not be set (written to 1) in the same write access. Writing the bits to 1 simultaneously can cause UNPREDICTABLE behavior.

For SYSRESETREQ, the reset is not guaranteed to take place immediately. A typical code sequence to synchronize reset following a write to the relevant control bit is:

```
      DSB;
Loop  B   Loop;
```

In addition the Application Interrupt and Reset Control register provides a mechanism, VECTCLRACTIVE to reset the active state of all exceptions. Writing 1 to the VECTCLRACTIVE bit clears the Active state of all exceptions and the Exception Number in the IPSR (see *The special-purpose program status registers (xPSR)* on page B1-8). Once complete, the IPSR and Active state of all exceptions will read as zero.

Note

This applies to Active state only, Pending state is not updated.

Reset and debug

Debug logic is fully reset by a Power-On Reset. Debug logic is only partially reset by a Local Reset. See *Debug and reset* on page C1-13 for details. Debuggers must only use VECTRESET when the core is halted, otherwise the effect is UNPREDICTABLE.

B1.5.17 Power management

ARMv7-M supports the use of Wait for Interrupt (`WFI`) and Wait for Event (`WFE`) instructions as part of a power management policy. Wait for Interrupt provides a mechanism for hardware to support entry to one or more sleep states. Hardware can suspend execution while waiting for a wakeup event. The levels of power saving and associated wakeup latency, while execution is suspended, are IMPLEMENTATION DEFINED.

Wait for Event provides a mechanism for software to suspend program execution with minimal or no impact on wakeup latency until a condition is met. Wait for Event allows some freedom for hardware to instigate power saving measures. Both `WFI` and `WFE` are hint instructions and can have no effect. They are generally used in software idle loops that resume program execution after an interrupt or event of interest has occurred.

Note

Code using `WFE` and `WFI` must handle spurious wakeup events as a result of a debug halt or other IMPLEMENTATION DEFINED reasons.

For more information, see:

- *Wait For Event and Send Event* on page B1-49
- *Wait For Interrupt* on page B1-51.

Where power management is supported, control and configuration is provided by the System Control Register, see *System Control Register (SCR)* on page B3-15. Support for the following features is provided:

- The transition of an interrupt from the Inactive to the Pending state can be configured as a wakeup event. This allows resumption of a Wait for Event instruction using a masked interrupt as the wakeup event.

- On an exception return, if no exceptions other than the returning exception are Active, there is a configuration bit to suspend execution. When the feature is enabled, the exception return is not performed. The subsequent activation of any exception behaves as a chained exception (see *Tail-chaining* on page B1-36).

  The suspended state can be exited spuriously. ARM recommends that software is written to handle spurious wakeup events and the associated exception return.

  Note

  ~~If PRIMASK is set and FAULTMASK is clear, an asynchronous exception which has a higher group priority than BASEPRI is not required by the architecture to result in the suspended state caused by SLEEPONEXIT being left. This is different from the treatment of `WFI` in this case.~~

- A qualifier that indicates support of different levels of sleep. The bit indicates that the wakeup time from the suspended execution can be longer than if the bit is not set. Typically this can be used to determine whether a PLL or other clock generator can be suspended. The exact behavior is IMPLEMENTATION DEFINED.

B1.5.18 Wait For Event and Send Event

ARMv7-M can support software-based synchronization with respect to system events using the `SEV` and `WFE` hint instructions. Software can:

- use the `WFE` instruction to indicate that it is able to suspend execution of a process or thread until an event occurs, permitting hardware to enter a low power state
- rely on a mechanism that is transparent to software and provides low latency wake up.

The Wait For Event system relies on hardware and software working together to achieve energy saving. For example, stalling execution of a processor until a device or another processor has set a flag:

- the hardware provides the mechanism to enter the Wait For Event low-power state
- software enters a polling loop to determine when the flag is set:
  - the polling processor issues a Wait For Event instruction as part of a polling loop if the flag is clear
  - an event is generated (hardware interrupt or Send Event instruction from another processor) when the flag is set.

The mechanism depends on the interaction of:

- WFE wake-up events, see *WFE wake-up events*
- the Event Register, see *The Event Register* on page B1-50
- the Send Event instruction, see *The Send Event instruction* on page B1-50
- the Wait For Event instruction, see *The Wait For Event instruction* on page B1-50.

WFE wake-up events

The following events are *WFE wake-up events*:

- the execution of an `SEV` instruction on any processor in the multiprocessor system

- any exception entering the Pending state if SEVONPEND in the System Control Register is set
- an asynchronous exception at a priority that pre-empts any currently active exceptions
- a debug event with debug enabled.

The Event Register

The Event Register is a single bit register for each processor in a multiprocessor system. When set, an Event Register indicates that an event has occurred, since the register was last cleared, that might prevent the processor needing to suspend operation on issuing a `WFE` instruction. The following conditions apply to the Event Register:

- ~~The value of the Event Register at reset is UNKNOWN.~~
- The Event Register is set by any `WFE` wake-up event or by the execution of an exception return instruction. For the definition of exception return instructions see *Exception return behavior* on page B1-25.
- The Event Register is only cleared by a `WFE` instruction.
- Software cannot read or write the value of the Event Register directly.

The Send Event instruction

The Send Event instruction, see *SEV* on page A6-212, causes a wake up event to be signaled to all processors in a multiprocessor system. The mechanism used to signal the event to the processors is IMPLEMENTATION DEFINED.

The Wait For Event instruction

The action of the Wait For Event instruction, see *WFE* on page A6-276, depends on the state of the Event Register:

- If the Event Register is set, the instruction clears the register and returns immediately.
- If the Event Register is clear the processor can suspend execution and enter a low-power state. It can remain in that state until the processor detects a WFE wake-up event or a reset. When the processor detects a WFE wake-up event, or earlier if the implementation chooses, the `WFE` instruction completes.

WFE wake up events can occur before a `WFE` instruction is issued. Software using the Wait For Event mechanism must be tolerant to spurious wake-up events, including multiple wake ups.

Pseudocode details of the Wait For Event lock mechanism

The `SetEventRegister()` pseudocode procedure sets the processor Event Register.

The `ClearEventRegister()` pseudocode procedure clears the processor Event Register.

The `EventRegistered()` pseudocode function returns TRUE if the processor Event Register is set and FALSE if it is clear:

```
boolean EventRegistered()
```

The `WaitForEvent()` pseudocode procedure optionally suspends execution until a WFE wake-up event or reset occurs, or until some earlier time if the implementation chooses. It is IMPLEMENTATION DEFINED whether restarting execution after the period of suspension causes a `ClearEventRegister()` to occur.

The `SendEvent()` pseudocode procedure sets the Event Register of every processor in a multiprocessor system.

B1.5.19 Wait For Interrupt

In ARMv7-M, Wait For Interrupt is supported through the hint instruction, `WFI`. For more information, see *WFI* on page A6-277.

When a processor issues a `WFI` instruction it can suspend execution and enter a low-power state. It can remain in that state until the processor detects a reset or one of the following *WFI wake-up events*:

- ~~an asynchronous exception at a priority that pre-empts any currently active exceptions~~

  ~~**Note**~~

  ~~If PRIMASK is set and FAULTMASK is clear, an asynchronous exception that has a higher group priority than any active exception and a higher group priority than BASEPRI results in a `WFI` instruction exit. If the group priority of the exception is less than or equal to the execution group priority, the exception is ignored.~~

- ~~a debug event with debug enabled.~~

~~When the hardware detects a WFI wake-up event, or earlier if the implementation chooses, the `WFI` instruction completes.~~

~~WFI wake-up events are recognized after the `WFI` instruction is issued.~~

~~**Note**~~

~~Because debug entry is one of the WFI wake-up events, ARM recommends that Wait For Interrupt is used as part of an idle loop rather than waiting for a single specific interrupt event to occur and then moving forward. This ensures the intervention of debug while waiting does not significantly change the function of the program being debugged.~~

Using WFI to indicate an idle state on bus interfaces

A common implementation practice is to complete any entry into power-down routines with a `WFI` instruction. Typically, the `WFI` instruction:

1. forces the suspension of execution, and of all associated bus activity
2. ceases to execute instructions from the processor.

The control logic required to do this typically tracks the activity of the bus interfaces of the processor. This means it can signal to an external power controller that there is no ongoing bus activity.

The exact nature of this interface is IMPLEMENTATION DEFINED, but the use of Wait For Interrupt as the only architecturally-defined mechanism that completely suspends execution makes it very suitable as the preferred power-down entry mechanism.

Pseudocode details of Wait For Interrupt

The `WaitForInterrupt()` pseudocode procedure optionally suspends execution until a WFI wake-up event or reset occurs, or until some earlier time if the implementation chooses.

Chapter B2
System Memory Model

This chapter contains information on the memory model pseudocode, the pseudocode associated with memory accesses. The chapter is made up of the following sections:

B2.1 Introduction

The pseudocode described in this chapter is associated with instruction fetches from memory and load or store data accesses.

The pseudocode hierarchy for a load or store instruction is as follows:

- the instruction operation uses the MemA[] or MemU[] helper function
- memory attributes are determined from the default system address map or using an MPU as defined in *The system address map* on page B3-2 or *Protected Memory System Architecture (PMSAv7)* on page B3-35 respectively.
- the access is governed by whether the access is a read or write, its address alignment, data endianness and memory attributes.

B2.2 Pseudocode details of general memory system operations

This section contains pseudocode describing general memory operations, in the subsections:

- *Memory data type definitions*.
- *Basic memory accesses* on page B2-4.
- *Interfaces to memory system specific pseudocode* on page B2-4.
- *Aligned memory accesses* on page B2-5
- *Unaligned memory accesses* on page B2-6
- *Reverse endianness* on page B2-7
- *Pseudocode details of operations on exclusive monitors* on page B2-8
- *Access permission checking* on page B2-10
- *MPU access control decode* on page B2-10
- *Default memory access decode* on page B2-11
- *MemManage fault handling* on page B2-13.

Additional pseudocode for memory protection is given in *MPU pseudocode* on page B3-36.

For a list of register names see Appendix I *Register Index*.
For a list of helper functions and procedures see *Miscellaneous helper procedures and functions* on page AppxG-22.

B2.2.1 Memory data type definitions

The following data type definitions are used by the memory system pseudocode functions:

```
// Types of memory

enumeration MemType {MemType_Normal, MemType_Device, MemType_StronglyOrdered};


// Memory attributes descriptor

type MemoryAttributes is (
    MemType type,
    bits(2) innerattrs,  // ’00’ = Non-cacheable; ’01’ = WBWA; ’10’ = WT; ’11’ = WBnWA
    bits(2) outerattrs,  // ’00’ = Non-cacheable; ’01’ = WBWA; ’10’ = WT; ’11’ = WBnWA
    boolean shareable
)



// Descriptor used to access the underlying memory array

type AddressDescriptor is (
    MemoryAttributes memattrs,
    bits(32)         physicaladdress
)
```

```
// Access permissions descriptor

type Permissions is (
    bits(3) ap,   // Access Permission bits
    bit     xn    // Execute Never bit
)
```

B2.2.2 Basic memory accesses

The _Mem[] function performs single-copy atomic, aligned, little-endian memory accesses to the underlying physical memory array of bytes:

```
bits(8*size) _Mem[AddressDescriptor memaddrdesc, integer size]           // non-assignment form
    assert size == 1 || size == 2 || size == 4;

_Mem[AddressDescriptor memaddrdesc, integer size] = bits(8*size) value   // assignment form
    assert size == 1 || size == 2 || size == 4;
```

The attributes in memaddrdesc.memattrs are used by the memory system to determine the memory type and ordering behaviors as described in *Memory types* on page A3-18 and *Memory access order* on page A3-30.

B2.2.3 Interfaces to memory system specific pseudocode

Global declarations are as follows:

```
boolean iswrite;            // TRUE for memory stores, FALSE for load accesses
boolean ispriv;             // TRUE if the instruction executing with privileged access
boolean isinstrfetch;       // TRUE if the memory access is associated with an instruction fetch
```

FindPriv() is used to determine if a privileged access. ValidateAddress() is used to resolve the memory attributes associated with an address and check the validity of the access where memory protection is enabled.

```
// FindPriv()
// ==========

boolean FindPriv()
    if (CurrentMode==Mode_Handler) OR ((CurrentMode==Mode_Thread)AND(CONTROL<0>=='0')) then
        ispriv = TRUE;
    else
        ispriv = FALSE;
    return ispriv;



ValidateAddress(bits(32) address, boolean ispriv, boolean iswrite, boolean isinstrfetch)
```

For more details on ValidateAddress(), see *MPU pseudocode* on page B3-36.

B2.2.4 Aligned memory accesses

The MemA[] function performs a memory access at the current privilege level, and the MemA_unpriv[] function performs an access that is always unprivileged. In both cases the architecture requires the access to be aligned, and generates an Alignment fault if it is not.

```
// MemA[]
// ======

bits(8*size) MemA[bits(32) address, integer size]
    return MemA_with_priv[address, size, FindPriv()];

MemA[bits(32) address, integer size] = bits(8*size) value
    MemA_with_priv[address, size, FindPriv()] = value;
    return;

// MemA_unpriv[]
// =============

bits(8*size) MemA_unpriv[bits(32) address, integer size]
    return MemA_with_priv[address, size, FALSE];

MemA_unpriv[bits(32) address, integer size] = bits(8*size) value
    MemA_with_priv[address, size, FALSE] = value;
    return;

~~// MemA_with_priv[]~~
~~// ================~~

~~// Non-assignment form~~

~~bits(8*size) MemA_with_priv[bits(32) address, integer size, boolean privileged]~~

    ~~// Sort out alignment~~
    ~~if address != Align(address, size) then~~
        ~~UFSR.UNALIGNED = '1';~~
        ~~ExceptionTaken(UsageFault);~~

    ~~// default address map or MPU~~
    ~~memaddrdesc = ValidateAddress(address, privileged, FALSE);~~

    ~~// Memory array access, and sort out endianness~~
    ~~value = _Mem[memaddrdesc, size];~~
    ~~if AIRCR.ENDIANESS == '1' then~~
        ~~value = BigEndianReverse(value, size);~~
    ~~return value;~~

~~// Assignment form~~

~~MemA_with_priv[bits(32) address, integer size, boolean privileged] = bits(8*size) value~~
```

```
    ~~// Sort out alignment~~
    ~~if address != Align(address, size) then~~
        ~~UFSR.UNALIGNED = '1';~~
        ~~ExceptionTaken(UsageFault);~~
    ~~// default address map or MPU~~
    ~~memaddrdesc = ValidateAddress(address, privileged, TRUE);~~

    ~~// Effect on exclusives~~
    ~~if memaddrdesc.memattrs.shareable then~~
        ~~ClearExclusiveByAddress(memaddrdesc.physicaladdress, ProcessorID(), size);~~

    ~~// Sort out endianness, then memory array access~~
    ~~if AIRCR.ENDIANESS == '1' then~~
        ~~value = BigEndianReverse(value, size);~~
    ~~_Mem[memaddrdesc,size] = value;~~

    ~~return;~~
```

B2.2.5 Unaligned memory accesses

The MemU[] function performs a memory access at the current privilege level, and the MemU_unpriv[] function performs an access that is always unprivileged.

In both cases:

- if the CCR.UNALIGN_TRP bit is 0, unaligned accesses are supported
- if the CCR.UNALIGN_TRP bit is 1, unaligned accesses produce Alignment faults.

```
// MemU[]
// ======

bits(8*size) MemU[bits(32) address, integer size]
    return MemU_with_priv[address, size, FindPriv()];

MemU[bits(32) address, integer size] = bits(8*size) value
    MemU_with_priv[address, size, FindPriv()] = value;
    return;

// MemU_unpriv[]
// =============

bits(8*size) MemU_unpriv[bits(32) address, integer size]
    return MemU_with_priv[address, size, FALSE];

MemU_unpriv[bits(32) address, integer size] = bits(8*size) value
    MemU_with_priv[address, size, FALSE] = value;
    return;

// MemU_with_priv[]
// ================
//
// Due to single-copy atomicity constraints, the aligned accesses are distinguished from
// the unaligned accesses:
```

```
// * aligned accesses are performed at their size
// * unaligned accesses are expressed as a set of bytes.

// Non-assignment form

bits(8*size) MemU_with_priv[bits(32) address, integer size, boolean privileged]

    bits(8*size) value;
    // Do aligned access, take alignment fault, or do sequence of bytes
    if address == Align(address, size) then
        value = MemA_with_priv[address, size, privileged];
    elsif CCR.UNALIGN_TRP == ’1’ then
        UFSR.UNALIGNED = ’1’;
        ExceptionTaken(UsageFault);
    else // if unaligned access
        for i = 0 to size-1
            value<8*i+7:8*i> = MemA_with_priv[address+i, 1, privileged];
        if AIRCR.~~ENDIANESS~~ == ’1’ then
            value = BigEndianReverse(value, size);

    return value;

// Assignment form

MemU_with_priv[bits(32) address, integer size, boolean privileged] = bits(8*size) value

    // Do aligned access, take alignment fault, or do sequence of bytes
    if address == Align(address, size) then
        MemA_with_priv[address, value, privileged] = value;
    elsif CCR.UNALIGN_TRP == ’1’ then
        UFSR.UNALIGNED = ’1’;
        ExceptionTaken(UsageFault);
    else // if unaligned access
        if AIRCR.~~ENDIANESS~~ == ’1’ then
            value = BigEndianReverse(value, size);
        for i = 0 to size-1
            MemA_with_priv[address+i, 1, privileged] = value<8*i+7:8*i>;

    return;
```

B2.2.6 Reverse endianness

The following pseudocode describes the operation to reverse endianness:

```
// BigEndianReverse()
// ==================

bits(8*N) BigEndianReverse (bits(8*N) value, integer N)
    assert N == 1 || N == 2 || N == 4;
    bits(8*N) result;
    case N of
        when 1
```

```
            result<7:0> = value<7:0>;
        when 2
            result<15:8> = value<7:0>;
            result<7:0> =  value<15:8>;
        when 4
            result<31:24> = value<7:0>;
            result<23:16> = value<15:8>;
            result<15:8> = value<23:16>;
            result<7:0> =  value<31:24>;
    return result;
```

B2.2.7 Pseudocode details of operations on exclusive monitors

The SetExclusiveMonitors() function sets the exclusive monitors for a load exclusive instruction. The ExclusiveMonitorsPass() function checks whether a store exclusive instruction still has possession of the exclusive monitors and therefore completes successfully.

```
// SetExclusiveMonitors()
// ======================

SetExclusiveMonitors(bits(32) address, integer size)

    memaddrdesc = ValidateAddress(address, FindPriv(), FALSE);

    if memaddrdesc.memattrs.shareable then
        MarkExclusiveGlobal(memaddrdesc.physicaladdress, ProcessorID(), size);

    MarkExclusiveLocal(memaddrdesc.physicaladdress, ProcessorID(), size);


// ExclusiveMonitorsPass()
// =======================

boolean ExclusiveMonitorsPass(bits(32) address, integer size)

    // It is IMPLEMENTATION DEFINED whether the detection of memory aborts happens
    // before or after the check on the local Exclusive Monitor. As a result a failure
    // of the local monitor can occur on some implementations even if the memory
    // access would give a memory abort.

    if address != Align(address, size) then
        UFSR.UNALIGNED = ’1’;
        ExceptionTaken(UsageFault);
    else
        memaddrdesc = ValidateAddress(address, FindPriv(), TRUE);

    passed = IsExclusiveLocal(memaddrdesc.physicaladdress, ProcessorID(), size);
    if memaddrdesc.memattrs.shareable then
        passed = passed && IsExclusiveGlobal(memaddrdesc.physicaladdress, ProcessorID(), size);
    if passed then
        ClearExclusiveLocal(ProcessorID());
    return passed;
```

The MarkExclusiveGlobal() procedure takes as arguments an address, the processor identifier processorid and the size of the transfer. The procedure records that processor processorid has requested exclusive access covering at least size bytes from the address. The size of region marked as exclusive is IMPLEMENTATION DEFINED, up to a limit of 2KB, and no smaller than size, and aligned in the address space to the size of the region. It is UNPREDICTABLE whether this causes any previous request for exclusive access to any other address by the same processor to be cleared.

```
MarkExclusiveGlobal(bits(32) address, integer processorid, integer size)
```

The `MarkExclusiveLocal()` procedure takes as arguments an address, the processor identifier `processorid` and the size of the transfer. The procedure records in a local record that processor `processorid` has requested exclusive access to an address covering at least `size` bytes from the address. The size of the region marked as exclusive is IMPLEMENTATION DEFINED, and can at its largest cover the whole of memory, but is no smaller than `size`, and is aligned in the address space to the size of the region. It is IMPLEMENTATION DEFINED whether this procedure also performs a `MarkExclusiveGlobal()` using the same parameters.

```
MarkExclusiveLocal(bits(32) address, integer processorid, integer size)
```

The `IsExclusiveGlobal()` function takes as arguments an address, the processor identifier `processorid` and the size of the transfer. The function returns TRUE if the processor `processorid` has marked in a global record an address range as exclusive access requested that covers at least the `size` bytes from the address. It is IMPLEMENTATION DEFINED whether it returns TRUE or FALSE if a global record has marked a different address as exclusive access requested. If no address is marked in a global record as exclusive access, `IsExclusiveGlobal()` returns FALSE.

```
boolean IsExclusiveGlobal(bits(32) address, integer processorid, integer size)
```

The `IsExclusiveLocal()` function takes as arguments an address, the processor identifier `processorid` and the size of the transfer. The function returns TRUE if the processor `processorid` has marked an address range as exclusive access requested that covers at least the `size` bytes from the address. It is IMPLEMENTATION DEFINED whether this function returns TRUE or FALSE if the address marked as exclusive access requested does not cover all of the `size` bytes from the address. If no address is marked as exclusive access requested, then this function returns FALSE. It is IMPLEMENTATION DEFINED whether this result is ANDed with the result of `IsExclusiveGlobal()` with the same parameters.

```
boolean IsExclusiveLocal(bits(32) address, integer processorid, integer size)
```

The `ClearExclusiveByAddress()` procedure takes as arguments an address, the processor identifier `processorid` and the size of the transfer. The procedure clears the global records of all processors, other than `processorid`, for which an address region including any of the `size` bytes starting from the supplied address has had a request for an exclusive access. It is IMPLEMENTATION DEFINED whether the equivalent global record of the processor `processorid` is also cleared if any of the `size` bytes starting from the address has had a request for an exclusive access, or if any other address has had a request for an exclusive access.

```
ClearExclusiveByAddress(bits(32) address, integer processorid, integer size)
```

The `ClearExclusiveLocal()` procedure takes the argument processor identifier `processorid`. The procedure clears the local record of processor `processorid` for which an address has had a request for an exclusive access. It is IMPLEMENTATION DEFINED whether this operation also clears the global record of processor `processorid` that an address has had a request for an exclusive access.

```
ClearExclusiveLocal(integer processorid)
```

B2.2.8 Access permission checking

The following pseudocode describes checking the access permission. Permissions are checked against access control information associated with a region when memory protection is supported and enabled, or against access control attributes associated with the default memory map.

```
// CheckPermission()
// =================

CheckPermission(Permissions perms, bits(32) address, boolean iswrite, boolean ispriv,
                boolean isinstrfetch)

    case perms.ap of
        when ’000’  fault = TRUE;
        when ’001’  fault = !ispriv;
        when ’010’  fault = !ispriv && iswrite;
        when ’011’  fault = FALSE;
        when ’100’  UNPREDICTABLE;
        when ’101’  fault = !ispriv || iswrite;
        when ’110’  fault = iswrite;
        when ’111’  fault = iswrite;
            else
                UNPREDICTABLE;

    if isinstrfetch then
        if fault || perms.xn then
            MMSR.IACCVIOL = ’1’;
            MMSR.MMARVALID = ’0’;
            ExceptionTaken(MemManage);
    elsif fault then
        MMSR.DACCVIOL = ’1’;
        MMAR = address;
        MMSR.MMARVALID = ’1’;
        ExceptionTaken(MemManage);
    return;
```

B2.2.9 MPU access control decode

The following pseudocode describes the memory attribute decode that is used when memory protection is enabled. See *MPU pseudocode* on page B3-36 for information on when `DefaultTEXDecode()` is called.

```
// DefaultTEXDecode()
// ==================

MemoryAttributes DefaultTEXDecode(bits(5) texcb, bit S)

    MemoryAttributes memattrs;

    case texcb of
```

```
        when '00000'
            memattrs.type = MemType_StronglyOrdered;
            memattrs.innerattrs = '00';  // Non-cacheable
            memattrs.outerattrs = '00';  // Non-cacheable
            memattrs.shareable = TRUE;
        when '00001'
            memattrs.type = MemType_Device;
            memattrs.innerattrs = '00';  // Non-cacheable
            memattrs.outerattrs = '00';  // Non-cacheable
            memattrs.shareable = TRUE;
        when "0001x", '00100'
            memattrs.type = MemType_Normal;
            memattrs.innerattrs = texcb<1:0>;
            memattrs.outerattrs = texcb<1:0>;
            memattrs.shareable = (S == '1');
        when '00110'
            IMPLEMENTATION_DEFINED setting of memattrs;
        when '00111'
            memattrs.type = MemType_Normal;
            memattrs.innerattrs = '01';  // Write-back write-allocate cacheable
            memattrs.outerattrs = '01';  // Write-back write-allocate cacheable
            memattrs.shareable = (S == '1');
        when '01000'
            memattrs.type = MemType_Device;
            memattrs.innerattrs = '00';  // Non-cacheable
            memattrs.outerattrs = '00';  // Non-cacheable
            memattrs.shareable = FALSE;
        when "1xxxx"
            memattrs.type = MemType_Normal;
            memattrs.innerattrs = texcb<1:0>;
            memattrs.outerattrs = texcb<3:2>;
            memattrs.shareable = (S == '1');
        otherwise
            UNPREDICTABLE;               // reserved cases

    return memattrs;
```

B2.2.10 Default memory access decode

The following pseudocode describes the default memory attribute decode, when memory protection is disabled, not supported, or cases where the protection control is overridden. See *MPU pseudocode* on page B3-36 for information on when `DefaultMemoryAttributes()` is called.

```
~~// DefaultMemoryAttributes()~~
~~// ========================~~


~~MemoryAttributes DefaultMemoryAttributes(bits(32) address, boolean isinstrfetch)~~
    ~~MemoryAttributes memattrs;~~

    ~~if (((UInt(PA<31:28>) > 9) OR (PA<31:29> == ’010’)) AND isinstrfetch) then~~
        ~~ExceptionTaken(HardFault);      // Instruction execution attempted from XN memory~~
```

```
~~case PA<31:29> of~~
    ~~when ‘000’~~
        ~~memattrs.type = MemType_Normal;~~
        ~~memattrs.innerattrs = ‘10’;~~
        ~~memattrs.shareable = ‘0’;~~

    ~~when ‘001’~~
        ~~memattrs.type = MemType_Normal;~~
        ~~memattrs.innerattrs = ‘01’;~~
        ~~memattrs.shareable = ‘0’;~~
    ~~when ‘010’~~
        ~~memattrs.type = MemType_Device;~~
        ~~memattrs.innerattrs = ‘00’;~~
        ~~memattrs.shareable = ‘0’;~~
    ~~when ‘011’~~
        ~~memattrs.type = MemType_Normal;~~
        ~~memattrs.innerattrs = ‘01’;~~
        ~~memattrs.shareable = ‘0’;~~
    ~~when ‘100’~~
        ~~memattrs.type = MemType_Normal;~~
        ~~memattrs.innerattrs = ‘10’;~~
        ~~memattrs.shareable = ‘0’;~~
    ~~when ‘101’~~
        ~~memattrs.type = MemType_Device;~~
        ~~memattrs.innerattrs = ‘00’;~~
        ~~memattrs.shareable = ‘1’;~~
    ~~when ‘110’~~
        ~~memattrs.type = MemType_Device;~~
        ~~memattrs.innerattrs = ‘00’;~~
        ~~memattrs.shareable = ‘0’;~~
    ~~when ‘111’~~
        ~~if PA<28:20> = ‘00000000’ then~~
            ~~memattrs.type = MemType_StronglyOrdered;~~
            ~~memattrs.innerattrs = ‘00’;~~
            ~~memattrs.shareable = ‘1’;~~
        ~~else~~
            ~~memattrs.type = MemType_Device;~~
            ~~memattrs.innerattrs = ‘00’;~~
            ~~memattrs.shareable = ‘0’;~~

    ~~// Outer attributes are the same as the inner attributes in all cases.~~
    ~~memattrs.outerattrs = memattrs.innerattrs;~~

~~return memattrs;~~
```

B2.2.11 MemManage fault handling

Memory access violations are reported as MemManage faults. If the fault is disabled, the fault will escalate to a HardFault exception. See *Overview of the exceptions supported* on page B1-14 and *Fault behavior* on page B1-39 for more information.

Chapter B3
System Address Map

This chapter contains information on the system address map. It contains the following sections:

B3.1 The system address map

For ARMv7-M, the 32-bit address space is predefined, with subdivision for code, data, and peripherals, as well as regions for on-chip (tightly coupled to the core) and off-chip resources.The address space supports 8 x 0.5GB primary partitions:

- Code
- SRAM
- Peripheral
- 2 x RAM regions
- 2 x Device regions
- System

Physical addresses are architecturally assigned for use as event entry points (vectors), system control, and configuration. The event entry points are all with respect to a table base address, where the base address is configured to an IMPLEMENTATION DEFINED value on reset, then maintained in an address space reserved for system configuration and control. To meet this and other system needs, the address space `0xE0000000` to `0xFFFFFFFF` is RESERVED for system level use.

Table B3-1 on page B3-3 describes the ARMv7-M default address map.

- XN refers to eXecute Never for the region and will fault (MemManage exception) any attempt to execute in the region.
- The Cache column indicates inner/outer cache policy to support system caches. The policy allows a declared cache type to be demoted but not promoted.

  WT: write through, can be treated as non-cached

  WBWA: write-back, write allocate, can be treated as write-through or non-cached
- Shared indicates to the system that the access is intended to support shared use from multiple agents; multiple processors and/or DMA agents within a coherent memory domain.
- It is IMPLEMENTATION DEFINED which portions of the overall address space are designated read-write, which are read-only (for example Flash memory), and which are no-access (unpopulated parts of the address map).
- An unaligned or multi-word access which crosses a 0.5GB address boundary is UNPREDICTABLE.

For additional information on memory attributes and the memory model see Chapter A2 *Application Level Programmers' Model*.

Table B3-1 ARMv7-M address map

	Name	Device Type	XN	Cache	Description

`0x00000000-`
`0x1FFFFFFF`
	Code	Normal	-	WT	Typically ROM or flash memory. Memory required from address 0x0 to support the vector table for system boot code on reset.

`0x20000000-`
`0x3FFFFFFF`
	SRAM	Normal	-	WBWA	SRAM region typically used for on-chip RAM.

`0x40000000-`
`0x5FFFFFFF`
	Peripheral	Device	XN	-	on-chip peripheral address space

`0x60000000-`
`0x7FFFFFFF`
	RAM	Normal	-	WBWA	memory with write-back, write allocate cache attribute for L2/L3 cache support

`0x80000000-`
`0x9FFFFFFF`
	RAM	Normal	-	WT	memory with write-thru cache attribute

`0xA0000000-`
`0xBFFFFFFF`
	Device	Device, shared	XN	-	shared device space

`0xC0000000-`
`0xDFFFFFFF`
	Device	Device	XN	-	non-shared device space

`0xE0000000-`
`0xFFFFFFFF`
	System	-	XN	-	system segment for the PPB and vendor system peripherals

+~~0000000~~	PPB	SO, (shared)	XN		1MB region reserved as a Private Peripheral Bus. The PPB supports key resources including the System Control Space, and debug features.
+~~0100000~~	Vendor_SYS	Device	XN		vendor system. It is suggested that vendor resources start at `0xF0000000` (~~+GB~~ offset).

To support a user (unprivileged) and supervisor (privileged) software model, a memory protection scheme is required to control the access rights. The Protected memory System Architecture for ARMv7-M (PMSAv7) is an optional system level feature described in *Protected Memory System Architecture (PMSAv7)* on page B3-35. An implementation of PMSAv7 is known as a Memory Protection Unit (MPU).

The address map described in Table B3-1 is the default map for an MPU when it is disabled, and the only address map supported when no MPU is present. The default map can be enabled as a background region for privileged accesses when the MPU is enabled. See the definition of PRIVDEFENA in *MPU Control Register (MPU_CTRL)* on page B3-40.

Note

When an MPU is enabled, the MPU is restricted in how it can change the default memory map attributes associated with System space (address `0xE0000000` or higher). System space is always marked as XN. System space which defaults to Device can be changed to Strongly-Ordered, but cannot be mapped to Normal memory. The PPB memory attributes cannot be remapped by an MPU.

B3.1.1 General rules applying to PPB register access

The Private Peripheral Bus (PPB), address range `0xE0000000` to `0xE0100000`, supports the following general rules:

- The region is defined as Strongly Ordered memory – see *Strongly-ordered memory* on page A3-25 and *Memory access restrictions* on page A3-26.
- Registers are always accessed little endian regardless of the endian state of the processor.
- In general, registers support word accesses only, with byte and halfword access UNPREDICTABLE. Several registers (namely priority and fault status registers) are a concatenation of byte aligned bit fields affecting different resources. In these cases, the registers[1] can be declared as 8-bit or 16-bit registers with an appropriate address offset within the 32-bit register base address.
- The term set means writing the value to 1, and the term clear(ed) means writing the value to 0. Where the term applies to multiple bits, all bits assume the written value.
- The term disable means writing the bit value to 0, the term enable means writing the bit value to 1.
- Where a bit is defined as clear on read, the following atomic behavior must be guaranteed when the bit is being read coincident with an event which sets the bit
  - If the bit reads as one, the bit is cleared by the read operation
  - If the bit reads as zero, the bit is set and read/cleared by a subsequent read operation
- A reserved register or bit field must be treated as UNK/SBZP.

Unprivileged (User) access to the PPB causes BusFault errors unless otherwise stated. Notable exceptions are:

- Unprivileged accesses can be enabled to the Software Trigger Interrupt Register in the System Control Space by programming a control bit in the Configuration Control Register.
- For debug related resources, see *General rules applying to debug register access* on page C1-6 for exception details.

[1] Registers only support byte and halfword access where it is explicitly stated in the register definition.

Note

The Flash Patch and Breakpoint block (FPB, see *Flash Patch and Breakpoint (FPB) support* on page C1-61) is designated a debug resource. Alternatively, FPB resources can be used as a means of updating software as part of a product maintenance policy. The address remapping behavior of the FPB is not specific to debug operation. Debug functionality is reduced when FPB resources are allocated to software maintenance.

B3.2 System Control Space (SCS)

The System Control Space is a memory-mapped 4kB address space which is used along with the special-purpose registers to provide arrays of 32-bit registers for configuration, status reporting and control. The SCS breaks down into the following groups:

- System Control/ID
- CPUID space
- System control, configuration and status
- Fault reporting
- A SysTick system timer
- A Nested Vectored Interrupt Controller (NVIC), supporting up to 496 discrete external interrupts. The NVIC shares a common prioritization model with the other exceptions.
- A Protected Memory System Architecture (PMSAv7) – see *Protected Memory System Architecture (PMSAv7)* on page B3-35
- System debug – see Chapter C1 *ARMv7-M Debug*

Table B3-2 defines the address space breakdown of the SCS register groups.

Table B3-2 SCS address space regions

System Control Space (address range `0xE000E000` to `0xE000EFFF`)		
Group	**Address Range(s)**	**Notes**
System Control/ID	`0xE000E000-0xE000E00F`	includes the Interrupt Controller Type and Auxiliary Control registers
	`0xE000ED00-0xE000ED8F`	System control block, includes the primary (CPUID) register
	`0xE000EF00-0xE000EF8F`	includes the SW Trigger Interrupt Register
	`0xE000EF90-0xE000EFCF`	IMPLEMENTATION DEFINED
	`0xE000EFD0-0xE000EFFF`	Microcontroller-specific ID space
SysTick	`0xE000E010-0xE000E0FF`	System Timer
NVIC	`0xE000E100-0xE000ECFF`	External interrupt controller
MPU	`0xE000ED90-0xE000EDEF`	Memory Protection Unit
Debug	`0xE000EDF0-0xE000EEFF`	Debug control and configuration

Detailed breakdown of the register groups is provided as part of the appropriate feature definition:

- System ID block in *System ID register support in the SCS* on page B3-10
- System control and configuration in *The System Control Block (SCB)* on page B3-10
- SysTick system timer in *System timer - SysTick* on page B3-24
- NVIC in *Nested Vectored Interrupt Controller (NVIC)* on page B3-28
- MPU in Chapter *Protected Memory System Architecture (PMSAv7)* on page B3-35
- Debug in Chapter C1 *ARMv7-M Debug*

B3.2.1 System control and ID blocks

System control and ID is supported by registers within subregions of the System Control Space as defined in Table B3-3.

Table B3-3 System control and ID registers

Address	Type	Reset Value	Name	Function
`0xE000E000`	R/W	`0x00000000`		Master Control register - RESERVED
`0xE000E004`	RO	IMPLEMENTATION DEFINED	ICTR	Interrupt Controller Type Register
`0xE000E008`	R/W	IMPLEMENTATION DEFINED	ACTLR	Auxiliary Control Register
…				…
`0xE000ED00`	RO	IMPLEMENTATION DEFINED	CPUID	CPUID Base Register
`0xE000ED04`	R/W	`0x00000000`	ICSR	Interrupt Control State Register
`0xE000ED08`	R/W	`0x00000000`	VTOR	Vector Table Offset Register
`0xE000ED0C`	R/W	bits [10:8] = 0b000	AIRCR	Application Interrupt/Reset Control Register
`0xE000ED10`	R/W	`0x00000000`	SCR	System Control Register
`0xE000ED14`	R/W	`0x00000000`	CCR	Configuration Control Register
`0xE000ED18`	R/W	`0x00000000`	SHPR1	System Handlers 4-7 Priority Register
`0xE000ED1C`	R/W	`0x00000000`	SHPR2	System Handlers 8-11 Priority Register
`0xE000ED20`	R/W	`0x00000000`	SHPR3	System Handlers 12-15 Priority Register
`0xE000ED24`	R/W	`0x00000000`	SHCSR	System Handler Control and State Register

Table B3-3 System control and ID registers (continued)

Address	Type	Reset Value	Name	Function
`0xE000ED28`	R/W	`0x00000000`	CFSR	Configurable Fault Status Register with separately accessible bit fields for: 1. MemManage fault status (MMFSR) 2. BusFault status (BFSR) 3. UsageFault status (UFSR)
`0xE000ED2C`	R/W	`0x00000000`	HFSR	HardFault Status Register
`0xE000ED30`	R/W	`0x00000000`[a]	DFSR	DebugFault Status Register
`0xE000ED34`	R/W	UNKNOWN	MMFAR	MemManage Address Register
`0xE000ED38`	R/W	UNKNOWN	BFAR	BusFault Address Register
`0xE000ED3C`	R/W	UNKNOWN	AFSR	Auxiliary Fault Status Register (IMPLEMENTATION DEFINED)

`0xE000ED40`
-
`0xE000ED7F`
				Reserved for CPUID Table (See Appendix A *CPUID*)

`0xE000ED88`	R/W	`0x00000000`	CPACR	Coprocessor Access Control Register
…				…

`0xE000EDF0`
-
`0xE000EEFF`
				See *Debug register support in the SCS* on page C1-19

`0xE000EF00`	WO		STIR	Software Trigger Interrupt Register
unused				reserved

`0xE000EF90`
-
`0xE000EFCF`
				IMPLEMENTATION DEFINED

Table B3-3 System control and ID registers (continued)

Address	Type	Reset Value	Name	Function
`0xE000EFD0`	RO	see notes	PID4	Peripheral identification register
`0xE000EFD4`	RO		PID5	
`0xE000EFD8`	RO		PID6	
`0xE000EFDC`	RO		PID7	
`0xE000EFE0`	RO		PID0	
`0xE000EFE4`	RO		PID1	
`0xE000EFE8`	RO		PID2	
`0xE000EFEC`	RO		PID3	
`0xE000EFF0`	RO		CID0	Component identification register
`0xE000EFF4`	RO		CID1	
`0xE000EFF8`	RO		CID2	
`0xE000EFFC`	RO		CID3	
Notes PIDx and CIDx: The ID registers should be CoreSight compatible or RAZ. See Appendix A *CPUID* for more information.				

a. Power-on reset only.

The Interrupt Controller Type Register (ICTR)

For details on the Interrupt Controller Type Register, see *Interrupt Controller Type Register – (0xE000E004)* on page B3-32.

The Auxiliary Control Register (ACTLR)

The Auxiliary Control Register, ACTLR, provides implementation-specific configuration and control options. The contents of this register are IMPLEMENTATION DEFINED.

Table B3-4 Auxiliary Control Register – (0xE000E008)

Bits	R/W	Name	Function
[31:0]			IMPLEMENTATION DEFINED

B3.2.2 System ID register support in the SCS

Support consists of the CPUID base register described in Table B3-5 and a block of ID attribute registers.

Table B3-5 CPUID Base Register – (CPUID, 0xE000ED00)

Bits	R/W	Name	Function
[31:24]	RO	IMPLEMENTER	Implementer code assigned by ARM. ARM == `0x41`.
[23:20]	RO	VARIANT	IMPLEMENTATION DEFINED
[19:16]	RO	(Constant)	Reads as `0xF`.
[15:4]	RO	PARTNO	IMPLEMENTATION DEFINED
[3:0]	RO	REVISION	IMPLEMENTATION DEFINED

The CPUID allows applications and debuggers to determine what kind of ARMv7-M processor they are using. The CPUID attribute ID registers can be read for specific details. This register is word accessible only.

CPU attribute ID registers

A bank of registers is defined as a series of 4-bit attribute fields. These attributes provide information details on the instruction set, memory model, and debug support present. See Appendix A *CPUID* for the detailed definition of the register support. These registers are word accessible only.

B3.2.3 The System Control Block (SCB)

Key control and status features of ARMv7-M are managed centrally in a System Control Block within the SCS. The SCB provides support for the following features:

- Software reset control at various levels
- Base address management (table pointer control) for the exception model
- System exception management (excludes external interrupts handled by the NVIC)
  - Exception enables
  - Setting or clearing exceptions to/from the pending state
  - Exception status (Inactive, Pending, or Active). Inactive is when an exception is neither Pending nor Active.
  - Priority setting (for configurable system exceptions)
  - Miscellaneous control and status information
- Priority grouping control – see *Priority grouping* on page B1-18 for a definition of priority grouping
- The exception (vector) number of the currently executing code and highest pending exception

- Miscellaneous control and status features including coprocessor access support
- Power management – sleep support.
- Fault status information – see *Fault behavior* on page B1-39 for an overview of fault handling
- Debug status information – supplemented with control and status in the debug specific register region. See Chapter C1 *ARMv7-M Debug* for debug details.

The SCB registers are listed in the following subsections in the order of their access address.

Interrupt Control State Register (ICSR)

Table B3-6 Interrupt Control and State Register – (0xE000ED04)

Bits	R/W	Name	Function
[31]	R/W	NMIPENDSET	Setting this bit will activate an NMI. Since NMI is higher priority than other exceptions, the NMI exception will activate as soon as it is registered.
[28]	R/W	PENDSVSET[a]	Set a pending PendSV interrupt. This is normally used to request a context switch. Reads back with current state (1 if Pending, 0 if not).
[27]	WO	PENDSVCLR	Clear a pending PendSV interrupt.
[26]	R/W	PENDSTSET[b]	Set a pending SysTick. Reads back with current state (1 if Pending, 0 if not).
[25]	WO	PENDSTCLR	Clear a pending SysTick (whether set here or by the timer hardware).
[23]	RO	ISRPREEMPT	If set, a pending exception will be serviced on exit from the debug halt state.
[22]	RO	ISRPENDING	Indicates if an external configurable (NVIC generated) interrupt is pending.
[20:12]	RO	VECTPENDING	Indicates the exception number for the highest priority pending exception. value == 0: no pending exceptions
[11]	RO	RETTOBASE	~~This bit is 1 when the set of all active exceptions minus the IPSR_current_exception yields the empty set.~~
[8:0]	RO	VECTACTIVE	value == 0: Thread mode value > 1: the exception number[c] for the current executing exception
Unused			Reserved

a. writing PENDSVSET and PENDSVCLR to '1' concurrently is UNPREDICTABLE

b. writing PENDSTSET and PENDSTCLR to '1' concurrently is UNPREDICTABLE

c. this is the same value as IPSR[8:0]

Vector Table Offset Register (VTOR)

Table B3-7 Vector Table Offset Register – (0xE000ED08)

Bits[a]	R/W	Name	Function
[29]	R/W	TBLBASE	value ==0: Table base is in CODE, base address `0x00000000` value ==1: Table base is in RAM, base address `0x20000000`
[28:7]	R/W	TBLOFF	Table offset address[28:7] from the base address defined by TBLBASE. The offset address is 32-word aligned. Where more than 16 external interrupts are used, the offset word alignment must be increased to accommodate vectors for all the exceptions and interrupts supported and keep the required table size naturally aligned. See *Interrupt Controller Type Register (ICTR)* on page B3-32 for information on the number of interrupts supported.
Unused			Reserved

[a] An implementation can include configuration input signals that determine the reset value of the TBLOFF bit field. If there is no configuration input signals to determine the reset value of this field then it resets to 0. A non-zero value must comply with the alignment restrictions described in *The vector table* on page B1-16. TBLBASE resets to zero.

~~An implementation can include configuration input signals that determine the reset value of the STKALIGN bit. If there is no configuration input signal to determine the reset value of this bit then it resets to 0.. A non-zero value must comply with the alignment restrictions described in *The vector table* on page B1-16.~~

Application Interrupt and Reset Control Register (AIRCR)

Table B3-8 Application Interrupt and Reset Control Register – (0xE000ED0C)

Bits	R/W	Name	Function
[31:16]	WO	VECTKEY	Vector Key. `0x05FA` must be written anytime this register is written, otherwise the write is ignored (no bits are changed in the register).
[31:16]	RO	VECTKEYSTAT	Reads as `0xFA05`.
[15]	RO	~~ENDIANESS~~	This bit is static or configured by a hardware input on reset. value == 0: little endian value == 1: big endian
[10:8]	R/W	PRIGROUP	Priority grouping position (binary point). This field sets the interpretation of priority registers, both for handlers and standard interrupts. For a definition of how this bit field is interpreted to assign priority and priority sub-group bit fields to the System Handler and NVIC priority registers, see *Priority grouping* on page B1-18) This field is cleared on reset.
[2]	WO	SYSRESETREQ	Writing this bit 1 will cause a signal to be asserted to the external system to indicate a reset is requested. The signal is required to generate a Local Reset. The bit self-clears as part of the reset sequence.

Table B3-8 Application Interrupt and Reset Control Register – (0xE000ED0C) (continued)

Bits	R/W	Name	Function
[1]	WO	VECTCLRACTIVE	Clears all active state information for fixed and configurable exceptions. This bit self-clears. Note: It is the application's responsibility to re-initialize the stack. Writing a 1 to this bit when not in debug state (halted) is UNPREDICTABLE.
[0]	WO	VECTRESET	Local system reset, see *Reset management* on page B1-47 for details. This bit self-clears. Writing a 1 to this bit when not in debug state (halted) is UNPREDICTABLE. If VECTRESET and SYSRESETREQ are set simultaneously, the behavior is UNPREDICTABLE.
Unused			Reserved

System Control Register (SCR)

Table B3-9 System Control Register (0xE000ED10)

Bits	R/W	Name	Function
[4]	R/W	SEVONPEND	When enabled, interrupt transitions from Inactive to Pending are included in the list of wakeup events for the `WFE` instruction. See *WFE wake-up events* on page B1-49 for more information.
[2]	R/W	SLEEPDEEP	A qualifying hint that indicates waking from sleep might take longer. Implementations can take advantage of the feature to identify a lower power sleep state.
[1]	R/W	SLEEPONEXIT	When set, the core can enter a sleep state on ~~exit from an ISR when it is returning to the Base level of activation~~.
Unused			Reserved

A debugger can read S_SLEEP (see *Debug Halting Control and Status Register (DHCSR)* on page C1-20) to detect if sleeping.

Configuration and Control Register (CCR)

Table B3-10 Configuration and Control Register (0xE000ED14)

Bits	R/W	Name	Function
[9]	~~R/W~~	STKALIGN	1: on exception entry, the SP used prior to the exception is adjusted to be 8-byte aligned and the context to restore it is saved. The SP is restored on the associated exception return. 0: only 4-byte alignment is guaranteed for the SP used prior to the exception on exception entry. ~~The bit is cleared on reset.~~ See *Stack alignment on exception entry* on page B1-24 for more information.
[8]	R/W	BFHFNMIGN	When enabled (=1), this causes handlers running at priority -1 and -2 to ignore Precise data access faults. When disabled (=0), these bus faults will cause a lock-up as explained in *Unrecoverable exception cases* on page B1-44.
[4]	R/W	DIV_0_TRP	Enable bit (=1) for trap on Divide by 0.
[3]	R/W	UNALIGN_TRP	Enable bit (=1) for trapping unaligned half or full word accesses[a]
[1]	R/W	USERSETMPEND	When this bit is set (=1), the core allows unprivileged (user) code to write the Software Trigger Interrupt register. See *Software Trigger Interrupt Register (STIR)* on page B3-23.
[0]	R/W	NONBASETHRDENA	0 (default): Thread state can only be entered at the Base level of activation (will fault if attempted to another level of activation). 1: Thread state can be entered at any level by controlled return value. See *Exception return behavior* on page B1-25 for details.
Unused			Reserved

[a] Unaligned load-store multiples and halfword/word exclusive accesses will always fault.

System Handler Priority Register 1 (SHPR1)

This register is byte, aligned halfword and word accessible.

Table B3-11 System Handler Priority Register 1 – (0xE000ED18)

Bits	R/W	Name	Function
[31:24]	R/W	PRI_7	Priority of system handler 7 - reserved
[23:16]	R/W	PRI_6	Priority of system handler 6 - UsageFault
[15:8]	R/W	PRI_5	Priority of system handler 5 - BusFault
[7:0]	R/W	PRI_4	Priority of system handler 4 - MemManage

System Handler Priority Register 2 (SHPR2)

This register is byte, aligned halfword and word accessible.

Table B3-12 System Handler Priority Register 2 – (0xE000ED1C)

Bits	R/W	Name	Function
[31:24]	R/W	PRI_11	Priority of system handler 11 - SVCall
[23:16]	R/W	PRI_10	Priority of system handler 10 - reserved
[15:8]	R/W	PRI_9	Priority of system handler 9 - reserved
[7:0]	R/W	PRI_8	Priority of system handler 8 - reserved

System Handler Priority Register 3 (SHPR3)

This register is byte, aligned halfword and word accessible.

Table B3-13 System Handler Priority Register 3 – (0xE000ED20)

Bits	R/W	Name	Function
[31:24]	R/W	PRI_15	Priority of system handler 15 - SysTick
[23:16]	R/W	PRI_14	Priority of system handler 14 - PendSV
[15:8]	R/W	PRI_13	Priority of system handler 13 - reserved
[7:0]	R/W	PRI_12	Priority of system handler 12 - DebugMonitor

System Handler Control and State Register (SHCSR)

Table B3-14 System Handler Control and State Register – (0xE000ED24)

Bits	R/W	Name	Function
[18]	R/W	USGFAULTENA	Enable for UsageFault.
[17]	R/W	BUSFAULTENA	Enable for BusFault.
[16]	R/W	MEMFAULTENA	Enable for MemManage fault.
[15]	R/W	SVCALLPENDED	Reads as 1 if SVCall is Pending (see note 1).
[14]	R/W	BUSFAULTPENDED	Reads as 1 if BusFault is Pending.
[13]	R/W	MEMFAULTPENDED	Reads as 1 if MemManage is Pending.
[12]	R/W	USGFAULTPENDED	Reads as 1 if UsageFault is Pending.
[11]	R/W	SYSTICKACT	Reads as 1 if SysTick is Active (see note 2).
[10]	R/W	PENDSVACT	Reads as 1 if PendSV is Active.
[8]	R/W	MONITORACT	Reads as 1 if the Monitor is Active.
[7]	R/W	SVCALLACT	Reads as 1 if SVCall is Active.
[3]	R/W	USGFAULTACT	Reads as 1 if UsageFault is Active.
[1]	R/W	BUSFAULTACT	Reads as 1 if BusFault is Active.
[0]	R/W	MEMFAULTACT	Reads as 1 if MemManage is Active.

Note 1: Pending state bits: A bit is set when the exception started to invoke, but was replaced by a higher priority exception

Note 2: Active state bits: the bit is set when the associated exception is the current exception or a nested (pre-empted) exception.

Configurable Fault Status Registers (UserFault, BusFault, and MemManage)

The 3 Configurable Fault Status Registers are one or two bytes each and packed into one word. The registers can be accessed collectively as a word as illustrated in Table B3-15 on page B3-1910-14, or individually as a byte or halfword:

- For the UsageFault Status Register, see Table B3-18 on page B3-20
- For the BusFault Status Register, see Table B3-17 on page B3-20
- For the MemManage Status Register, see Table B3-16 on page B3-19

The Configurable Fault Status Registers are byte, aligned halfword and word accessible.

Table B3-15 Configurable Fault Status Registers (CFSR, 0xE000ED28)

Bits	R/W	Fault Status register	Function
[31:16]	R/W1C	UsageFault	Provides information on UsageFault exceptions
[15:8]	R/W1C	BusFault	Provides information on BusFault exceptions
[7:0]	R/W1C	MemManage	Provides information on MemManage exceptions
R/W1C: Read/Write-one-to-clear			

The MemManage Status Register contains the status of MPU faults.

Table B3-16 MemManage Status Register (MMFSR, 0x~~E000D28~~)

Bits	R/W	Name	Function
[7]	R/W1C	MMARVALID	This bit is set if the MMAR register has valid contents.
[4]	R/W1C	MSTKERR	A derived MemManage fault has occurred on exception entry.
[3]	R/W1C	MUNSTKERR	A derived MemManage fault has occurred on exception return.
[1]	R/W1C	DACCVIOL	Data access violation. The MMAR is set to the data address which the load/store tried to access.
[0]	R/W1C	IACCVIOL	MPU or eXecuteNever (XN) default memory map access violation on an instruction fetch. The fault is only signalled if the instruction is issued.
		Unused bits	Reserved
R/W1C: Read/Write-one-to-clear			

The BusFault Status Register contains the status of bus errors resulting from instruction prefetches and data accesses.

Table B3-17 BusFault Status Register (BFSR, 0xE000ED29)

Bits	R/W	Name	Function
[7]	R/W1C	BFARVALID	This bit is set if the BFAR register has valid contents.
[4]	R/W1C	STKERR	This bit indicates a derived bus fault has occurred on exception entry.
[3]	R/W1C	UNSTKERR	This bit indicates a derived bus fault has occurred on exception return.
[2]	R/W1C	IMPRECISERR	Imprecise data access error.
[1]	R/W1C	PRECISERR	Precise data access error. The BFAR is written with the faulting address.
[0]	R/W1C	IBUSERR	This bit indicates a bus fault on an instruction prefetch. The fault is only signalled if the instruction is issued.
		Unused bits	Reserved
R/W1C: Read/Write-one-to-clear			

The UsageFault Status Register contains the status of a variety of instruction execution and data access faults.

Table B3-18 UsageFault Status Register (UFSR, 0xE000ED2A)

Bits	R/W	Name	Function
[15:10]			Reserved
[9]	R/W1C	DIVBYZERO	Divide by zero error. When `SDIV` or `UDIV` instruction is used with a divisor of 0, this fault will occur if DIV_0_TRP is enabled.
[8]	R/W1C	UNALIGNED	Unaligned access error. Multi-word accesses always fault if not word aligned. Unaligned word and halfwords can be configured to fault (UNALIGN_TRP is enabled)
[7:4]			Reserved
[3]	R/W1C	NOCP	Coprocessor access error (the coprocessor is disabled or not present)
[2]	R/W1C	INVPC	Integrity check error on EXC_RETURN.

Table B3-18 UsageFault Status Register (UFSR, 0xE000ED2A) (continued)

Bits	R/W	Name	Function
[1]	R/W1C	INVSTATE	Invalid EPSR.T bit or illegal EPSR.IT bits for executing instruction
[0]	R/W1C	UNDEFINSTR	~~Undefined~~ instruction executed (including those associated with an enabled Coprocessor).
R/W1C: Read/Write-one-to-clear			

The UsageFault bits are additive; that is, if more than one fault occurs, all associated bits are set.

HardFault Status register (HFSR)

The HardFault Status Register contains a set of bits which indicate causes of faults.

Table B3-19 HardFault Status Register (0xE000ED2C)

Bits	R/W	Name	Function
[31]	R/W1C	DEBUGEVT	Debug event, and the Debug Fault Status Register has been updated. Only set when halting debug is disabled (C_DEBUGEN = 0) See *Debug event behavior* on page C1-14 for more information
[30]	R/W1C	FORCED	Configurable fault cannot be activated due to priority or because it is disabled. Priority escalated to a HardFault exception. See *Priority escalation* on page B1-19.
[1]	R/W1C	VECTTBL	Fault was due to vector table read on exception processing.
		Not used	Reserved
R/W1C: Read/Write-one-to-clear			

Debug Fault Status Register (DFSR)

See Chapter C1 *ARMv7-M Debug* for a full description of debug support within ARMv7-M and *Debug Fault Status Register (DFSR)* on page C1-19 for the Debug Fault Status Register.

Auxiliary Fault Status Register (AFSR)

The register is located at address 0xE000ED3C. The contents of this register are system specific and IMPLEMENTATION DEFINED.

MemManage Address Register (MMFAR)

Table B3-20 MemManage Address Register (0xE000ED34)

Bits	R/W	Name	Function
[31:0]	R/W	ADDRESS	Data address MPU faulted. This is the location which a load or store attempted to access which was faulted. The MemManage Status Register provides the cause, and indicates if the content of this register is valid.When an unaligned access faults, the address will be the actual address which faulted; since an access may be split into multiple parts (each aligned), this address therefore may be any offset in the range of the requested size.

BusFault Address Register (BFAR)

Table B3-21 BusFault Address Register (0xE000ED38)

Bits	R/W	Name	Function
[31:0]	R/W	ADDRESS	Updated on precise data access faults. The value is the faulting address associated with the attempted access. The BusFault Status Register provides information on the reason, and indicates if the content of this register is valid.For unaligned access faults, the address is the address requested by the instruction, which is not necessarily the address which faulted.

Coprocessor Access Control Register (CPACR)

Table B3-22 Coprocessor Access Control Register– (0xE000ED88)

Bits	R/W	Name	Function
[~~31~~:0]	R/W	CPACR	Each bit pair corresponds to a coprocessor (bits [1:0] assigned to CP0, …, bits [~~31:30~~] assigned to CP~~15~~). The interpretation of each bit pair is as follows: 0b00: Access denied – generates a NOCP UsageFault 0b01: Privileged access only. User access will generate a NOCP fault. 0b10: Reserved (UNPREDICTABLE). 0b11: Full access.

If a bit pair is written with a 0b01 or 0b11 and it reads back as 0b00, the coprocessor is not fitted

Software Trigger Interrupt Register (STIR)

Table B3-23 Software Trigger Interrupt Register – (0xE000EF00)

Bits	R/W	Name	Function
[~~9:0~~]	WO	INTID	The value written in this field is the interrupt to be triggered. This acts the same as storing to the corresponding ~~ISPR[x]~~ set-pending NVIC register bit. See *Interrupt Set-Pending and Clear-Pending Registers (NVIC_ISPRx and NVIC_ICPRx)* on page B3-33. This register applies to external interrupts only. The value written is (ExceptionNumber - 16), see *Exception number definition* on page B1-16.

B3.3 System timer - SysTick

ARMv7-M includes an architected system timer – SysTick.

SysTick provides a simple, 24-bit clear-on-write, decrementing, wrap-on-zero counter with a flexible control mechanism. The counter can be used in several different ways, by example:

- An RTOS tick timer which fires at a programmable rate (for example 100Hz) and invokes a SysTick routine.
- A high speed alarm timer using Core clock.
- A variable rate alarm or signal timer – the duration range dependent on the reference clock used and the dynamic range of the counter.
- A simple counter. Software can use this to measure time to completion and time used.
- An internal clock source control based on missing/meeting durations. The COUNTFLAG bit-field in the control and status register can be used to determine if an action completed within a set duration, as part of a dynamic clock management control loop.

B3.3.1 Theory of operation

The timer consists of four registers:

- A control and status counter to configure its clock, enable the counter, enable the SysTick interrupt, and determine counter status.
- The reload value for the counter, used to provide the counter's wrap value.
- The current value of the counter.
- A calibration value register, indicating the preload value necessary for a 10ms (100Hz) system clock.

When enabled, the timer will count down from the ~~reload value~~ to zero, reload (wrap) to the value in the SysTick Reload Value Register on the next clock edge, then decrement on subsequent clocks. Writing a value of zero to the Reload Value Register disables the counter on the next wrap. When the counter transitions to zero, the COUNTFLAG status bit is set. The COUNTFLAG bit clears on reads.

~~Writing to the Current Value Register~~ will clear the register and the COUNTFLAG status bit. The write does not trigger the SysTick exception logic. On a read, the current value is the value of the register at the time the register is accessed.

The calibration value TENMS allows software to scale the counter to other desired clock rates within the counter's dynamic range.

If the core is in Debug state (halted), the counter will not decrement.

The timer is clocked with respect to a reference clock. The reference clock can be the core clock or an external clock source. Where an external clock source is used, the implementation must document the relationship between the core clock and the external reference. This is required for system timing calibration, taking account of metastability, clock skew and jitter.

B3.3.2 System timer register support in the SCS

Table B3-24 summarizes the register support provided within the SCS address map.

Table B3-24 SysTick register support in the SCS

Address	R/W	Reset Value	Name	Function
`0xE000E010`	R/W	0x00000000	SYST_CSR	SysTick Control and Status
`0xE000E014`	R/W	~~Unpredictable~~	SYST_RVR	SysTick Reload value
`0xE000E018`	R/W	~~Unpredictable~~	SYST_CVR	SysTick Current value
`0xE000E01C`	RO	IMP DEF	SYST_CALIB	SysTick Calibration value
…to `0xE000E0FF`				Reserved

Descriptions of the four SysTick registers are provided in the following subsections.

SysTick Control and Status Register (SYST_CSR)

Table B3-25 SysTick Control and Status Register – (0xE000E010)

Bits	R/W	Name	Function
[16]	RO	COUNTFLAG	Returns 1 if timer counted to 0 since last time this register was read. COUNTFLAG is set by a count transition from 1 => 0. COUNTFLAG is cleared on read or by a write to the Current Value register.
[2]	R/W	CLKSOURCE	0: clock source is (optional) external reference clock 1: core clock used for SysTick If no external clock provided, this bit will read as 1 and ignore writes.
[1]	R/W	TICKINT	If 1, counting down to 0 will cause the SysTick exception to be pended. Clearing the SysTick Current Value register by a register write in software will not cause SysTick to be pended.
[0]	R/W	ENABLE	0: the counter is disabled 1: the counter will operate in a multi-shot manner.
Unused			Reserved

SysTick Reload Value Register (SYST_RVR)

Table B3-26 SysTick Reload Value Register – (0xE000E014)

Bits	R/W	Name	Function
[31:24]	RAZ/WI		
[23:0]	R/W	RELOAD	Value to load into the Current Value register when the counter reaches 0.

SysTick Current Value Register (SYST_CVR)

Table B3-27 SysTick Current Value Register – (0xE000E018)

Bits	R/W	Name	Function
[31:0]	R/W	CURRENT	Current counter value. This is the value of the counter at the time it is sampled. The counter does not provide read-modify-write protection.The register is write-clear. A software write of any value will clear the register to 0. Unsupported bits RAZ (see SysTick Reload Value register).

SysTick Calibration value Register (SYST_CALIB)

Table B3-28 SysTick Calibration Value Register – (0xE000E01C)

Bits	R/W	Name	Function
[31]	RO	NOREF	If reads as 1, the Reference clock is not provided – the CLKSOURCE bit of the SysTick Control and Status register will be forced to 1 and cannot be cleared to 0.
[30]	RO	SKEW	If reads as 1, the calibration value for 10ms is inexact (due to clock frequency).
[29:24]			Reserved
[23:0]	RO	TENMS	An optional Reload value to be used for 10ms (100Hz) timing, subject to system clock skew errors. If the value reads as 0, the calibration value is not known.

B3.4 Nested Vectored Interrupt Controller (NVIC)

ARMv7-M provides an interrupt controller as an integral part of the ARMv7-M exception model. The interrupt controller operation aligns with ARM's General Interrupt Controller (GIC) specification, promoted for use with other architecture variants and ARMv7 profiles.

The ARMv7-M NVIC architecture supports up to 496 (IRQ[495:0]) discrete interrupts. The number of external interrupt lines supported can be determined from the read-only Interrupt Controller Type Register (ICTR) accessed at address 0xE000E004 in the System Control Space. See *Interrupt Controller Type Register (ICTR)* on page B3-32 for the register detail. The general registers associated with the NVIC are all accessible from a block of memory in the System Control Space as described in Table B3-29 on page B3-30.

B3.4.1 Theory of operation

ARMv7-M supports level-sensitive and pulse-sensitive interrupt behavior. This means that both level-sensitive and pulse-sensitive interrupts can be handled. Pulse interrupt sources must be held long enough to be sampled reliably by the core clock to ensure they are latched and become Pending. A subsequent pulse can re-pend the interrupt while it is Active, however, multiple pulses which occur during the Active period will only register as a single event for interrupt scheduling.

In summary:

- Pulses held for a clock period will act like edge-sensitive interrupts. These can re-pend when the interrupt is Active.

  Note

  A pulse must be cleared before the assertion of AIRCR.VECTCLRACTIVE or the associated exception return, otherwise the interrupt signal behaves as a level-sensitive input and the pending bit is asserted again.

- Level based interrupts will pend and activate the interrupt. The Interrupt Service Routine (ISR) can then access the peripheral, causing the level to be de-asserted. If the interrupt is still asserted on return from the interrupt, it will be pended again.

All NVIC interrupts have a programmable priority value and an associated exception number as part of the ARMv7-M exception model and its prioritization policy.

The NVIC supports the following features:

- NVIC interrupts can be enabled and disabled by writing to their corresponding Interrupt Set-Enable or Interrupt Clear-Enable register bit-field. The registers use a write-1-to-enable and write-1-to-clear policy, both registers reading back the current enabled state of the corresponding (32) interrupts.

  When an interrupt is disabled, interrupt assertion will cause the interrupt to become Pending, however, the interrupt will not activate. If an interrupt is Active when it is disabled, it remains in its Active state until cleared by reset or an exception return. Clearing the enable bit prevents new activations of the associated interrupt.

Interrupt enable bits can be hard-wired to zero where the associated interrupt line does not exist, or hard-wired to one where the associated interrupt line cannot be disabled.

- NVIC interrupts can be pended/un-pended using a complementary pair of registers to those used to enable/disable the interrupts, named the Set-Pending Register and Clear-Pending Register respectively. The registers use a write-1-to-enable and write-1-to-clear policy, both registers reading back the current pended state of the corresponding (32) interrupts. The Clear-Pending Register has no effect on the execution status of an Active interrupt.

  It is IMPLEMENTATION DEFINED for each interrupt line supported, whether an interrupt supports setting and/or clearing of the associated pend bit under software control.

- Active bit status is provided to allow software to determine whether an interrupt is Inactive, Active, Pending, or Active and Pending.

- NVIC interrupts are prioritized by updating an 8-bit field within a 32-bit register (each register supporting four interrupts). Priorities are maintained according to the ARMv7-M prioritization scheme. See *Exception priorities and pre-emption* on page B1-17.

In addition to an external hardware event or setting the appropriate bit in the Set-Pending registers, an external interrupt can be pended from software by writing its interrupt number (ExceptionNumber - 16) to the Software Trigger Interrupt Register described in *Software Trigger Interrupt Register (STIR)* on page B3-23.

External interrupt input behavior

The following pseudocode describes the relationship between external interrupt inputs and the NVIC behavior:

```
// DEFINITIONS

NVIC[] is an array of active high external interrupt input signals;
    // the type of signal (level or pulse) and its assertion level/sense is IMPLEMENTATION DEFINED
    // and might not be the same for all inputs

boolean Edge(integer INTNUM);    // Returns true if on a clock edge NVIC[INTNUM]
                                 // has changed from ’0’ to ’1’
boolean NVIC_Pending[INTNUM];    // an array of pending status bits for the external interrupts
integer INTNUM;                  // the external interrupt number


    // The WriteToRegField helper function returns TRUE on a write of ’1’ event
    // to the field FieldNumber of the RegName register.

boolean WriteToRegField(register RegName, integer FieldNumber)


boolean ExceptionIN(integer INTNUM);   // returns TRUE if exception entry in progress
                                       // to activate INTNUM
boolean ExceptionOUT(integer INTNUM);  // returns TRUE if exception return in progress
                                       // from active INTNUM
// INTERRUPT INTERFACE

sampleInterruptHi = WriteToRegField(AIRCR, VECTCLRACTIVE) || ExceptionOUT(INTNUM);
sampleInterruptLo = WriteToRegField(ICPR, INTNUM);

InterruptAssertion   =  Edge(INTNUM) || (NVIC[INTNUM] && sampleInterruptHi);
InterruptDeassertion =  !NVIC[INTNUM] && sampleInterruptLo;



// NVIC BEHAVIOR

clearPend = ExceptionIN(INTNUM) || InterruptDeassertion;
setPend   = InterruptAssertion || WriteToRegField(ISPR, INTNUM);

if clearPend && setPend then
    IMPLEMENTATION DEFINED whether NVIC_Pending[INTNUM] is TRUE or FALSE;
else
    NVIC_Pending[INTNUM] = setPend || (NVIC_Pending[INTNUM] && !clearPend);
```

B3.4.2 NVIC register support in the SCS

The Interrupt Controller Type Register (ICTR) and Software Trigger Interrupt Register (STIR) reside in the system control region. The ICTR is a read-only register which provides information on the number of external interrupts supported by the implementation. The STIR is a write-only register which enables pending of external interrupts by software. The system control region also includes status and configuration registers which apply to the NVIC as part of the general exception model.

All other external interrupt specific registers reside within the NVIC region of the SCS. Table B3-29 summarizes the NVIC specific registers in the SCS.

Table B3-29 NVIC register support in the SCS

Address	Type	Reset Value	Name	Function
0xE000E004	RO	IMPLEMENTATION DEFINED	ICTR	Interrupt Controller Type Register
0xE000EF00	WO	-	STIR	Software Trigger Interrupt Register
0xE000E100	R/W	0x00000000	NVIC_ISER0	Irq 0 to 31 Set-Enable Register
…	…	…		…

Table B3-29 NVIC register support in the SCS (continued)

Address	Type	Reset Value	Name	Function
0xE000E13C	R/W	0x00000000	NVIC_ISER15	Irq 480 to 495 Set-Enable Register[a]
0xE000E180	R/W	0x00000000	NVIC_ICER0	Irq 0 to 31 Clear-Enable Register
…	…	…		…
0xE000E1BC	R/W	0x00000000	NVIC_ICER15	Irq 480 to 495 Clear-Enable Register[a]
0xE000E200	R/W	0x00000000	NVIC_ISPR0	Irq 0 to 31 Set-Pending Register
…	…	…		…
0xE000E23C	R/W	0x00000000	NVIC_ISPR15	Irq 480 to 495 Set-Pending Register[a]
0xE000E280	R/W	0x00000000	NVIC_ICPR0	Irq 0 to 31 Clear-Pending Register
…	…	…		…
0xE000E2BC	RO	0x00000000	NVIC_ICPR15	Irq 480 to 495 Clear-Pending Register[a]
0xE000E300	RO	0x00000000	NVIC_IABR0	Irq 0 to 31 Active Bit Register
…	…	…		…
0xE000E37C	RO	0x00000000	NVIC_IABR15	Irq 480 to 495 Active Bit Register[a]
380-3FC	reserved			
0xE000E400	R/W	0x00000000	NVIC_IPR0	Irq 0 to 3 Priority Register
…	…	…		…
0xE000E7EC	R/W	0x00000000	NVIC_IPR123	Irq 492 to 495 Priority Register

Table B3-29 NVIC register support in the SCS (continued)

Address	Type	Reset Value	Name	Function
0xE000E7F0	reserved			
…	…	…		…
0xE000ECFF	reserved			

a. bits [31:16] reserved

Interrupt Controller Type Register (ICTR)

Table B3-30 Interrupt Controller Type Register – (0xE000E004)

Bits	R/W	Name	Function
[31:4]			Reserved
[3:0]	RO	INTLINESNUM	Number of interrupt lines supported by NVIC in granularities of 32.

INTLINESNUM is the total number of interrupt lines supported by an implementation, defined in groups of 32. This is encoded as follows:

- 0b0000: up to 32 interrupt lines supported
  (These interrupts are banked in a multi-processor system)
- 0b0001: up to 64 interrupt lines supported.
- 0b0010: up to 96 interrupt lines supported
- 0b0011: up to 128 interrupt lines supported
- …
- 0b1110: up to 480 interrupt lines supported
- 0b1111: up to 496 interrupt lines supported (a maximum of sixteen supported by this value).

INTLINESNUM can be used to determine which sets of registers in the NVIC register map are populated. NVIC register space access outside of the regions defined by INTLINESNUM is reserved.

Interrupt Set-Enable and Clear-Enable Registers (NVIC_ISERx and NVIC_ICERx)

Table B3-31 Interrupt Set-Enable Registers – (0xE000E100-E17C)

Bits	R/W	Name	Function
[31:0]	R/W	SETENA	Enable one or more interrupts within a group of 32. Each bit represents an interrupt number from N to N+31 (starting at interrupt 0, 32, 64, etc). Writing a 1 will enable the associated interrupt. Writing a 0 has no effect. The register reads back with the current enable state.

Table B3-32 Interrupt Clear-Enable Registers – (0xE000E180-E1FC)

Bits	R/W	Name	Function
[31:0]	R/W	CLRENA	Disable one or more interrupts within a group of 32. Each bit represents an interrupt number from N to N+31 (starting at interrupt 0, 32, 64, etc). Writing a 1 will disable the associated interrupt. Writing a 0 has no effect. The register reads back with the current enable state.

Interrupt Set-Pending and Clear-Pending Registers (NVIC_ISPRx and NVIC_ICPRx)

Table B3-33 Interrupt Set-Pending Registers – (0xE000E200-E27C)

Bits	R/W	Name	Function
[31:0]	R/W	SETPEND	Writing a 1 to a bit pends the associated interrupt under software control. Each bit represents an interrupt number from N to N+31 (starting at interrupt 0, 32, 64, etc). Writing a 0 to a bit has no effect on the associated interrupt.The register reads back with the current pending state.

Table B3-34 Interrupt Clear-Pending Registers – (0xE000E280-E2FC)

Bits	R/W	Name	Function
[31:0]	R/W	CLRPEND	Writing a 1 to a bit un-pends the associated interrupt under software control. Each bit represents an interrupt number from N to N+31 (starting at interrupt 0, 32, 64, etc). Writing a 0 to a bit has no effect on the associated interrupt. The register reads back with the current pending state.

Active Bit Register (NVIC_IABRx)

Table B3-35 Interrupt Active Bit Registers – (0xE000E300-E37C)

Bits	R/W	Name	Function
[31:0]	RO	ACTIVE	Each bit represents the current active state for the associated interrupt within a group of 32. Each bit represents an interrupt number from N to N+31 (starting at interrupt 0, 32, 64, etc).

Interrupt Priority Register (NVIC_IPRx)

These registers are byte, aligned halfword and word accessible.

Table B3-36 Interrupt Priority Registers – (0xE000E400-E7F8)

Bits	R/W	Name	Function
[31:24]	R/W	PRI_N3	Priority of interrupt number N+3 (3, 7, 11, etc).
[23:16]	R/W	PRI_N2	Priority of interrupt number N+2 (2, 6, 10, etc).
[15:8]	R/W	PRI_N1	Priority of interrupt number N+1 (1, 5, 9, etc).
[7:0]	R/W	PRI_N	Priority of interrupt number N (0, 4, 8, etc).

B3.5 Protected Memory System Architecture (PMSAv7)

To support a user (unprivileged) and supervisor (privileged) software model, a memory protection scheme is required to control the access rights. ARMv7-M supports the Protected Memory System Architecture (PMSAv7). The system address space of a PMSAv7 compliant system is protected by a Memory Protection Unit (MPU). The protected memory is divided up into a set of regions, with the number of regions supported IMPLEMENTATION DEFINED. While PMSAv7 supports region sizes as low as 32 bytes, finite register resources for the 4GB address space make the scheme inherently a coarse-grained protection scheme. The protection scheme is 100% predictive with all control information maintained in registers closely-coupled to the core. Memory accesses are only required for software control of the MPU register interface, see *Register support for PMSAv7 in the SCS* on page B3-38.

MPU support in ARMv7-M is optional, and co-exists with the system memory map described in *The system address map* on page B3-2 as follows:

- MPU support provides access right control on physical addresses. No address translation occurs in the MPU.
- When the MPU is disabled or not present, the system adopts the default system memory map listed in Table B3-1 on page B3-3.When the MPU is enabled, the enabled regions are used to define the system address map with the following provisos:
  - accesses to the Private Peripheral Bus (PPB) always uses the default system address map
  - exception vector reads from the Vector Address Table always use the default system address map
  - the MPU is restricted in how it can change the default memory map attributes associated with System space (address 0xE0000000 or higher).

    System space is always marked as XN (eXecute Never).

    ~~System space which defaults to Device can be changed to Strongly-Ordered, but cannot be mapped to Normal memory.~~
  - exceptions executing at a priority < 0 (NMI, HardFault, and exception handlers with FAULTMASK set) can be configured to run with the MPU enabled or disabled
  - the default system memory map can be configured to provide a background region for privileged accesses
  - accesses with an address match in more than one region use the highest matching region number for the access attributes
  - accesses which do not match all access conditions of a region address match (with the MPU enabled) or a background/default memory map match generate a fault.

B3.5.1 PMSAv7 compliant MPU operation

ARMv7-M only supports a unified memory model with respect to MPU region support. All enabled regions provide support for instruction and data accesses.

The base address, size and attributes of a region are all configurable, with the general rule that all regions are naturally aligned. This can be stated as:

```
RegionBaseAddress[(N-1):0] = 0, where N is log2(SizeofRegion_in_bytes)
```

Memory regions can vary in size as a power of 2. The supported sizes are 2^N, where $5 \le N \le 32$. Where there is an overlap between two regions, the register with the highest region number takes priority.

Sub-region support

For regions of 256 bytes or larger, the region can be divided up into eight sub-regions of size $2^{(N-3)}$. Sub-regions within a region can be disabled on an individual basis (8 disable bits) with respect to the associated region attribute register. When a sub-region is disabled, an access match is required from another region, or background matching if enabled. If an access match does not occur a fault is generated. Region sizes below 256 bytes do not support sub-regions. The sub-region disable field is SBZ/UNP for regions of less than 256 bytes in size.

ARMv7-M specific support

ARMv7-M supports the standard PMSAv7 memory model, plus the following extensions:

- An optimized two register update model, where the region being updated can be selected by writing to the MPU Region Base Address Register. This optimization applies to the first sixteen memory regions (0 ≤ RegionNumber ≤ 0xF) only.
- The MPU Region Base Address Register and the MPU Region Attribute and Size Register pairs are aliased in three consecutive dual-word locations. Using the two register update model, up to four regions can be modified by writing the appropriate even number of words using a single `STM` multi-word store instruction.

MPU pseudocode

The following pseudocode defines the operation of an ARMv7-M MPU. The terms used align with the MPU register names and bit field names described in *Register support for PMSAv7 in the SCS* on page B3-38.

```
~~// ValidateAddress()~~
~~// ================~~

~~AddressDescriptor ValidateAddressP(bits(32) address, boolean ispriv, boolean iswrite,~~
                                    ~~boolean isinstrfetch)~~

    ~~boolean isvectortablelookup;~~        ~~// TRUE if address associated with exception entry~~
    ~~boolean isPPBaccess;~~                ~~// TRUE if the address is to the PPB region~~
    ~~AddressDescriptor result;~~
    ~~Permissions perms;~~


    ~~// PMSA only supports shared (insruction and data) regions in ARMv7-M.~~

    ~~result.physicaladdress = address;~~
```

```
~~if isvectortablelookup OR isPPBaccess then       // bypass MPU for these accesses~~
    ~~result.memattrs = DefaultAttrs(PA);~~

~~elsif ((ICSR.VECTACTIVE == 2) || (ICSR.VECTACTIVE == 3)) then  // NMI, HardFault, or FAULTMASK~~
    ~~if MPUCR.HFNMIENA == ‘1’ then              // MPU enabled for NMI, HardFault and FAULTMASK~~
        ~~if MPUCR.ENABLE == ‘0’ then~~
            ~~UNPREDICTABLE;~~
    ~~else~~
        ~~result.memattrs = DefaultMemoryAttributes(PA);  // MPU disabled for NMI etc.~~

~~elsif MPUCR.ENABLE == ‘0’ then                          // MPU disabled~~
    ~~result.memattrs = DefaultMemoryAttributes(PA);~~
~~else                                                    // MPU is enabled~~

    ~~// Scan through regions looking for matching ones. If found, the last~~
    ~~// one matched is used.~~
    ~~region_found = FALSE;~~

    ~~for r=0 to MPUTR.DRegion-1~~
        ~~bits(16) size_enable    = MPURASR[r]<15:0>;~~
        ~~bits(32) base_address   = MPURBAR[r];~~
        ~~bits(16) access_control = MPURASR[r]<31:16>;~~

        ~~if size_enable<0> == ’1’ then  // Region is enabled~~
            ~~lsbit = UInt(size_enable<5:1>) + 1;~~
            ~~if lsbit < 2 then UNPREDICTABLE;~~

            ~~if lsbit == 32 || address<31:lsbit> == base_address<31:lsbit> then~~
                ~~if lsbit >= 8 then  // can have subregions~~
                    ~~subregion = UInt(address<lsbit-1:lsbit-3>);~~
                    ~~hit = (size_enable<subregion+8> == ’0’);~~
                ~~else~~
                    ~~hit = TRUE;~~

                ~~if hit then~~
                    ~~texcb = access_control<5:3,1:0>;~~
                    ~~S = access_control<2>;~~
                    ~~perms.ap = access_control<10:8>;~~
                    ~~perms.xn = access_control<12>;~~
                    ~~region_found = TRUE;~~

    ~~// Generate the memory attributes, and also the permissions if no region found.~~
    ~~if region_found then~~
        ~~result.memattrs = DefaultTEXDecode(texcb, S);~~
    ~~else~~
        ~~if MPUCR.PRIVDEFENA == ’0’ then~~
            ~~if isinstrfetch then~~
                ~~MMSR.IACCVIOL = ’1’;~~
                ~~MMSR.MMARVALID = ’0’;~~
                ~~ExceptionTaken(MemManage);~~
            ~~else~~
                ~~MMSR.DACCVIOL = ’1’;~~
                ~~MMAR = address;~~
                ~~MMSR.MMARVALID = ’1’;~~
                ~~ExceptionTaken(MemManage);~~
        ~~else~~
            ~~result.memattrs = DefaultMemoryAttributes(address);~~
            ~~perms.ap = '001';~~
            ~~perms.xn = if((UInt(PA<31:28>) > 9) OR (PA<31:29> == '010'));~~

        ~~// Check the permissions.~~
    ~~CheckPermission(perms, address, iswrite, ispriv, isinstrfetch);~~

~~return result;~~
```

MPU fault support

Instruction or data access violations cause a MemManage exception to be generated. ~~In the pseudocode, these are described by the pseudo-function MemManageFault()~~. See *Fault behavior* on page B1-39 for more details of MemManage exceptions.

B3.5.2 Register support for PMSAv7 in the SCS

Table B3-37 on page B3-39 summarizes the register support for a memory Protection Unit (MPU) in the System Control Space. In common with the general policy, all registers are byte, halfword, and word accessible unless stated otherwise. All MPU register addresses are mapped as little endian.

MPU registers require privileged memory accesses for reads and writes. Unprivileged (User) accesses generate ~~MemManage~~ faults.

There are three general MPU registers:

- The MPU Type Register specified in *MPU Type Register (MPU_TYPE)* on page B3-39. This register can be used to determine if an MPU exists, and the number of regions supported.
- The MPU Control Register specified in *MPU Control Register (MPU_CTRL)* on page B3-40. The MPU Control register includes a global enable bit which must be set to enable the MPU feature.
- The MPU Region Number Register specified in *MPU Region Number Register (MPU_RNR)* on page B3-41.

The MPU Region Number Register selects the associated region registers:

- The MPU Region Base Address Register specified in *MPU Region Base Address Register (MPU_RBAR)* on page B3-41.
- The MPU Region Attribute and Size Register to control the region size, sub-region access, access permissions, memory type and other properties of the memory region in *MPU Region Attribute and Size Register (MPU_RASR)* on page B3-42.

Each set of region registers contains its own region enable bit.

Where PMSAv7 is not supported, only the MPU Type Register is mandatory. The MPU Control Register is RAZ/WI, and all other registers in this region are reserved (UNK/SBZP).

Table B3-37 MPU register support in the SCS

Address	Type	Reset Value	Name	Function
`0xE000ED90`	RO	IMPLEMENTATION DEFINED	MPU_TYPE	MPU Type Register
`0xE000ED94`	R/W	0x00000000	MPU_CTRL	MPU Control Register
`0xE000ED98`	R/W	UNKNOWN	MPU_RNR	MPU Region Number Register
`0xE000ED9C`	R/W	UNKNOWN	MPU_RBAR	MPU Region Base Address Register
`0xE000EDA0`	R/W	UNKNOWN	MPU_RASR	MPU Region Attribute and Size Register
`0xE000EDA4`	R/W		MPU_RBAR_A1	MPU Alias 1~~: MPURBAR(RegionNumber+1)~~
`0xE000EDA8`	R/W		MPU_RASR_A1	MPU Alias 1~~: MPURASR(RegionNumber+1)~~
`0xE000EDAC`	R/W		MPU_RBAR_A2	MPU Alias 2~~: MPURBAR(RegionNumber+2)~~
`0xE000EDB0`	R/W		MPU_RASR_A2	MPU Alias 2~~: MPURASR(RegionNumber+2)~~
`0xE000EDB4`	R/W		MPU_RBAR_A3	MPU Alias 3~~: MPURBAR(RegionNumber+3)~~
`0xE000EDB8`	R/W		MPU_RASR_A3	MPU Alias 3~~: MPURASR(RegionNumber+3)~~
…to `0xE000EDEF`	…			Reserved for future protection registers

MPU Type Register (MPU_TYPE)

The MPU Type Register indicates how many regions are supported by the MPU. It can be used to determine if there is an MPU supported.

Table B3-38 MPU Type Register – (0xE000ED90)

Bits	R/W	Name	Function
[31:24]	-		reserved
[23:16]	RO	IREGION	RAZ. ARMv7-M only supports a unified MPU.
[15:8]	RO	DREGION	Specifies the number of regions supported by the MPU. If RAZ, the MPU is not supported.
[0]	RO	SEPARATE	RAZ. ARMv7-M only supports a unified MPU.

MPU Control Register (MPU_CTRL)

The MPU Control Register is used to enable the MPU. The register is cleared on reset. If no regions are enabled and the PRIVDEFENA and ENABLE bits are set, only privileged code can execute from the system address map.

Table B3-39 MPU Control Register – (0xE000ED94)

Bits	R/W	Name	Function
[31:3]	-	-	reserved
[2]	R/W	PRIVDEFENA	When the bit is set along with the ENABLE bit, the Default memory map (as defined in *The system address map* on page B3-2) is enabled as a background region for privileged access. The background region acts as though it were region number -1. MPU configured regions will override (take priority over) the default memory map. When the bit is clear, the default map is disabled. Instruction or data accesses not covered by a region will fault. When the ENABLE bit is clear, PRIVDEFENA is ignored.
[1]	R/W	HFNMIENA	~~When set along with the ENABLE bit, the MPU is enabled for HardFault, NMI, and exception handlers with FAULTMASK set.~~ When clear, the MPU will be disabled when in these handlers (regardless of the value of ENABLE). When HFNMIENA is set and ENABLE is clear, the system behavior is UNPREDICTABLE.
[0]	R/W	ENABLE	When set, the MPU is enabled. When clear, the MPU is disabled and the default memory map applies to privileged and User code.

If the MPU is not fitted, this register is RAZ/WI.

MPU Region Number Register (MPU_RNR)

The MPU Region Number Register is written to select the region to read or write.

Table B3-40 MPU Region Number Register – (0xE000ED98)

Bits	R/W	Name	Function
[31:8]	-	-	reserved
[7:0]	R/W	REGION	This field selects the region specific ~~MPURBAR~~ and ~~MPURASR~~. The ~~MPURNR~~ must be written before accessing the associated register pair, unless the alternative access feature (see ~~MPURBAR~~.VALID description) is used on a write access.

MPU Region Base Address Register (MPU_RBAR)

Table B3-41 MPU Region Base Address Register – (0xE000ED9C)

Bits	R/W	Name	Function
[31:~~N~~]	R/W	ADDR	Base address of the region. ~~The base address is naturally aligned according to the size of the region. N is defined as log2(SizeofRegion_in_bytes), where the size can be determined from the MPU Region Size Register.~~
~~[N-1:5]~~	~~R/W~~		~~RAZ/WI~~
[4]	R/W	VALID	When the bit is written with ’1’, the REGION field of this register is zero extended and copied into the ~~MPURNR.~~ When the bit is written with ’0’, ~~MPURNR~~ does not change. This bit is RAZ.
[3:0]	R/W	REGION	For register writes, see VALID bit field description. For register reads, the bit field reads as bits [3:0] of the ~~MPURNR~~.

The minimum size of region supported by an MPU Region Base Address Register is IMPLEMENTATION DEFINED. It can be determined by writing all 1’s to the region’s ~~MPURBAR~~[31:5] and reading back the bits set. The minimum size is determined from the number of trailing zeroes in the bit field. All regions support all size values from the minimum supported to 4GB (see SIZE in *MPU Region Attribute and Size Register – (0xE000EDA0)* on page B3-42). This register is UNPREDICTABLE if accessed other than as a word.

MPU Region Attribute and Size Register (MPU_RASR)

The Region Attribute and Size Register (~~MPURASR~~) defines the size and access behavior of the associated memory region. The register fields are described in Table B3-42.

Table B3-42 MPU Region Attribute and Size Register – (0xE000EDA0)

Bits	R/W	Name	Function
[31:16]	R/W	ATTRS	The MPU Region Attribute Register as defined in *Region attribute control* on page B3-43
[15:8]	R/W	SRD	For regions of 256 bytes or larger, the Sub-Region Disable bits are used to disable 1/8 of the region per bit. For any bit that is set, the relevant 1/8 of the region's address range is disabled with respect to the attribute settings.
[7:6]	-		reserved
[5:1]	R/W	SIZE	The region size as defined in Table B3-43.
[0]	R/W	ENABLE	When set, the associated region is enabled within the MPU. The global MPU enable bit must also be set for it to take effect.

Writing a SIZE field with a value greater than that supported by the associated MPU Region Base Address Register is UNPREDICTABLE.

The 8 Sub-Region Disable (SRD) bits allow the 1/8th sub-regions to be disabled individually. The least significant bit affects the 1/8th sub-region with the lowest address range, the most significant bit affects the highest addressed 1/8th sub-region. The sub-region disable bits are UNPREDICTABLE for region sizes of 32, 64, and 128 bytes. For further information on sub-region support, see *Sub-region support* on page B3-36.

Additional information on the SIZE and ATTRS fields is provided below.

Region size control

The size of a region is encoded in the MPU Region Size Register as shown in Table B3-43.

Table B3-43 Region Size Encoding

Size Encoding	Region size in bytes	Size Encoding	Region size in bytes
00000	-	10000	128KB
00001	-	10001	256KB
00010	-	10010	512KB
00011	-	10011	1MB

Table B3-43 Region Size Encoding (continued)

Size Encoding	Region size in bytes	Size Encoding	Region size in bytes
00100	32	10100	2MB
00101	64	10101	4MB
00110	128	10110	8MB
00111	256	10111	16MB
01000	512	11000	32MB
01001	1KB	11001	64MB
01010	2KB	11010	128MB
01011	4KB	11011	256MB
01100	8KB	11100	512MB
01101	16KB	11101	1GB
01110	32KB	11110	2GB
01111	64KB	11111	4GB

Region attribute control

TheMPU region attribute fields define the memory type, cache, and access policies for a given memory region.

Table B3-44 Region attribute fields

31 29	28	27	26 24	23 22	21 19	18	17	16
Reserved	XN	Reserved	AP	Reserved	TEX	S	C	B

The TEX, S, C and B bits control the device type and cacheability as shown in Table B3-45. Cache information is exported by the core on instruction fetches and data accesses. The information can be used to support system caches on a system bus.

Table B3-45 TEX/CB/S Encoding

TEX	C	B	Description	Memory Type	Region Shareable?
000	0	0	Strongly ordered	Strongly ordered	Shareable
000	0	1	Shared device	Device	Shareable
000	1	0	Outer and inner write through, no write allocate	Normal	s[a]
000	1	1	Outer and inner write back, no write allocate	Normal	s
001	0	0	Outer and inner Non-cacheable	Normal	s
001	0	1	RESERVED	RESERVED	RESERVED
001	1	0	IMPLEMENTATION DEFINED	IMPLEMENTATION DEFINED	IMPLEMENTATION DEFINED
001	1	1	Outer and inner write back; write and read allocate	Normal	s
010	0	0	Non-shared device	Device	Not shareable
010	0	1	RESERVED	RESERVED	RESERVED
010	1	X	RESERVED	RESERVED	RESERVED
011	X	X	RESERVED	RESERVED	RESERVED
1BB	A	A	Cached memory. BB = outer policy[b], AA == inner policy See Table B3-46 on page B3-45 for encoding details.	Normal	s

[a] "s" is the S bit (bit [2]) from the MPU Region Attribute Register

[b] the cache policy refers to cache rules to be exported on the bus

Table B3-46 Cache policy encoding

Memory Attribute Encoding (AA and BB)	Cache Policy
00	Non-cacheable
01	Write back, write and read allocate
10	Write through, no write allocate
11	Write back, no write allocate

The AP bits, AP[2:0], are used for access permissions. These are shown in Table B3-47.

Table B3-47 AP encoding

AP[2:0]	Privileged Permissions	User Permissions	Description
000	No Access	No Access	All accesses generate a permission fault
001	Read/Write	No Access	Privileged access only
010	Read/Write	Read Only	Unprivileged (User) writes generate permission faults
011	Read/Write	Read/Write	Full access
100	UNPREDICTABLE	UNPREDICTABLE	RESERVED
101	Read Only	No Access	Privileged read only
110	Read Only	Read Only	Privileged/User read only
111	Read Only	Read Only	Privileged/User read only

The XN bit provides an eXecute Never capability. Instructions must have read access as defined by the AP bits and XN clear for correct execution, otherwise a MemManage fault is generated when the instruction is issued.

Table B3-48 XN encoding

XN	Description
0	Instruction fetches allowed
1	Instruction fetches not allowed

MPU alias register support

The MPU Region Base Address Register and the MPU Region Attribute and Size Register form a pair of words in the address range of 0xE000ED9C to 0xE000EDA3. Three aliases of this address range are provided at addresses 0xE000D9C + 8, +16, and +24 as shown in Table B3-37 on page B3-39. Using the register aliases along with the REGION/VALID fields of the MPU Region Base Address Register, software can efficiently update the four regions with a stream of word writes, assuming all the regions accessed are in the range Region0 to Region15.

Chapter B4
ARMv7-M System Instructions

As previously stated, ARMv7-M only executes instructions in Thumb state. The full list of supported instructions is provided in *Alphabetical list of ARMv7-M Thumb instructions* on page A6-17. To support reading and writing the special-purpose registers under software control, ARMv7-M provides three system instructions:

`CPS`

`MRS`

`MSR`

B4.1 Alphabetical list of ARMv7-M system instructions

The ARMv7-M system instructions are defined in this section:

- *CPS*
- *MRS* on page B4-4
- *MSR (register)* on page B4-8[1]

B4.1.1 CPS

Change Processor State changes one or more of the special-purpose register PRIMASK and FAULTMASK values.

Encoding T1 ARMv6-M, ARMv7-M

Enhanced functionality in ARMv7-M.

Not allowed in IT block.

`CPS<effect> <iflags>`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	0	1	1	0	0	1	1	im	(0)	(0)	I	F

```
enable = (im == '0');  disable = (im == '1');
affectPRI = (I == '1');  affectFAULT = (F == '1');
if InITBlock() then UNPREDICTABLE;
```

Assembler syntax

`CPS<effect><q> <iflags>`

where:

`<effect>` Specifies the effect required on PRIMASK and FAULTMASK. This is one of:

`IE` Interrupt Enable. This sets the specified bits to 0.

`ID` Interrupt Disable. This sets the specified bits to 1.

`<q>` See *Standard assembler syntax fields* on page A6-7. A `CPS` instruction must be unconditional.

`<iflags>` Is a sequence of one or more of the following, specifying which masks are affected:

`i` PRIMASK. Raises the ~~current~~ priority to 0 when set to 1. This is a 1-bit register, which supports privileged access only.

`f` FAULTMASK. Raises the ~~current~~ priority to -1 (the same as HardFault) when it is set to 1. ~~Thisis~~ a 1-bit register, which can only be set by privileged code with a lower priority than -1. The register self-clears on return from any exception other than NMI.

[1] MSR(immediate) is a valid instruction in other ARMv7 profiles and earlier architecture variants. The MSR (immediate) encoding is UNDEFINED in ARMv7-M.

Operation

```
EncodingSpecificOperations();
if CurrentModeIsPrivileged() then
    if enable then
        if affectPRI then PRIMASK<0> = '0';
        if affectFAULT then FAULTMASK<0> = '0';
    if disable then
        if affectPRI then PRIMASK<0> = '1';
        if affectFAULT && ExecutionPriority > -1 then FAULTMASK<0> = '1';
```

Exceptions

None.

Notes

Privilege Any unprivileged (User) code attempt to write the masks is ignored.

Masks and CPS

The `CPSIE` and `CPSID` instructions are equivalent to using an `MSR` instruction:

- The `CPSIE i` instruction is equivalent to writing a 0 into PRIMASK
- The `CPSID i` instruction is equivalent to writing a 1 into PRIMASK
- The `CPSIE f` instruction is equivalent to writing a 0 into FAULTMASK
- The `CPSID f` instruction is equivalent to writing a 1 into FAULTMASK.

B4.1.2 MRS

Move to Register from Special Register moves the value from the selected special-purpose register into a general-purpose register.

Encoding T1 ARMv6-M, ARMv7-M Enhanced functionality in ARMv7-M.

```
MRS<c> <Rd>,<spec_reg>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	1	1	(0)	(1)	(1)	(1)	(1)	1	0	(0)	0	Rd				SYSm							

```
d = UInt(Rd);
if d IN {13,15} || !(UInt(SYSm) IN {0..3,5..9,16..20}) then UNPREDICTABLE;
```

Assembler syntax

```
MRS<c><q>  <Rd>, <spec_reg>
```

where:

`<c><q>` See *Standard assembler syntax fields* on page A6-7.

`<Rd>` Specifies the destination register.

`<spec_reg>` Encoded in SYSm, specifies one of the following:

Special register	Contents	SYSm value
APSR	The flags from previous instructions	0
IAPSR	A composite of IPSR and APSR	1
EAPSR	A composite of EPSR and APSR	2
XPSR	A composite of all three PSR registers	3
IPSR	The Interrupt status register	5
EPSR	The execution status register	6
IEPSR	A composite of IPSR and EPSR	7
MSP	The Main Stack pointer	8
PSP	The Process Stack pointer	9
PRIMASK	Register to mask out configurable exceptions	16 [a]
BASEPRI	The base priority register	17 [b]

Add footnote:

The EPSR is RAZ

Special register	Contents	SYSm value
BASEPRI_MAX	This acts as an alias of BASEPRI on reads	18 [c]
FAULTMASK	Register to raise priority to the HardFault level	19 [d]
CONTROL	The special-purpose control register	20 [e]
RSVD	RESERVED	unused

[a] Raises the ~~current~~ priority to 0 when set to 1. This is a 1-bit register.

[b] Changes the current pre-emption priority mask to a value between 0 and N. 0 means the mask is disabled. The register only has an effect when the value (1 to N) is lower (higher priority) than the non-masked priority level of the executing instruction stream.
The register can have up to 8 bits (depending on the number of priorities supported), and it is formatted exactly the same as other priority registers.
The register is affected by the PRIGROUP (binary point) field. See *Exception priorities and pre-emption* on page B1-17 for more details. Only the pre-emption part of the priority is used by BASEPRI for masking.

[c] When used with the MSR instruction, it performs a conditional write.

[d] This register raises the ~~current~~ priority to -1 (the same as HardFault) when it is set to 1. This can only be set by privileged code with a priority below -1 (not NMI or HardFault), and self-clears on return from any exception other than NMI. This is a 1-bit register.

[e] The control register is composed of the following bits:
[0] = Thread mode privilege: 0 means privileged, 1 means unprivileged (User). This bit resets to 0.
[1] = Current stack pointer: 0 is Main stack (MSP), 1 is alternate stack (PSP if Thread mode, RESERVED if Handler mode). This bit resets to 0.

Operation

```
if ConditionPassed() then
    R[d] = 0;
    case  SYSm<7:3> of
        when '00000'
            if SYSm<0> == '1' and CurrentModeIsPrivileged() then
                R[d]<8:0> = IPSR<8:0>;
            if SYSm<1> == '1' then
                R[d]<26:24> = '000';     /* EPSR reads as zero */
                R[d]<15:10> = '000000';
            if SYSm<2> == '0' then
                R[d]<31:27> = APSR<31:27>;
        when '00001'
            if CurrentModeIsPrivileged() then
                case SYSm<2:0> of
                    when '000'
                        R[d] = MSP;
                    when '001'
                        R[d] = PSP;
        when '00010'
            case SYSm<2:0> of
                when '000'
                    R[d]<0> = if CurrentModeIsPrivileged() then
                        PRIMASK<0> else '0';
                when '001'
                    R[d]<7:0> = if CurrentModeIsPrivileged() then
                        BASEPRI<7:0> else '00000000';
                when '010'
                    R[d]<7:0> = if CurrentModeIsPrivileged() then
                        BASEPRI<7:0> else '00000000';
                when '011'
                    R[d]<0> = if CurrentModeIsPrivileged() then
                        FAULTMASK<0> else '0';
                when ‘100‘
                    R[d]<1:0> = CONTROL<1:0>;
```

Exceptions

None.

Notes

Privilege If User code attempts to read any stack pointer or the IPSR, it returns 0s.

EPSR None of the EPSR bits are readable during normal execution. They all read as 0 when read using MRS (Halting debug can read them via the register transfer mechanism).

Bit positions The PSR bit positions are defined in *The special-purpose program status registers (xPSR)* on page B1-8.

B4.1.3 MSR (register)

Move to Special Register from ARM Register moves the value of a general-purpose register to the selected special-purpose register.

Encoding T1 ARMv6-M, ARMv7-M Enhanced functionality in ARMv7-M.

```
MSR<c> <spec_reg>,<Rn>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	0	0	(0)	Rn				1	0	(0)	0	(1)	(0)	(0)	(0)	SYSm							

```
n = UInt(Rn);
if n IN {13,15} || !(UInt(SYSm) IN {0..3,5..9,16..20}) then UNPREDICTABLE;
```

Assembler syntax

```
MSR<c><q>  <spec_reg>, <Rn>
```

where:

`<c><q>` See *Standard assembler syntax fields* on page A6-7.

`<Rn>` Is the general-purpose register to receive the special register contents.

`<spec_reg>` Encoded in SYSm, specifies one of the following:

Special register	Contents	SYSm value
APSR	The flags from previous instructions	0
IAPSR	A composite of IPSR and APSR	1
EAPSR	A composite of EPSR and APSR	2
XPSR	A composite of all three PSR registers	3
IPSR	The Interrupt status register	5
EPSR	The execution status register ~~(reads as zero, see Notes)~~	6
IEPSR	A composite of IPSR and EPSR	7
MSP	The Main Stack pointer	8
PSP	The Process Stack pointer	9
PRIMASK	Register to mask out configurable exceptions	16 [a]
BASEPRI	The base priority register	17 [b]

Insert footnote:

The EPSR ignores writes

Special register	Contents	SYSm value
BASEPRI_MAX	On writes, raises BASEPRI but does not lower it	18 [c]
FAULTMASK	Register to raise priority to the HardFault level	19 [d]
CONTROL	The special-purpose control register	20 [e]
RSVD	RESERVED	unused

[a] Raises the ~~current~~ priority to 0 when set to 1. This is a 1-bit register.

[b] Changes the current pre-emption priority mask to a value between 0 and N. 0 means the mask is disabled. The register only has an effect when the value (1 to N) is lower (higher priority) than the non-masked priority level of the executing instruction stream.
The register can have up to 8 bits (depending on the number of priorities supported), and it is formatted exactly the same as other priority registers.
The register is affected by the PRIGROUP (binary point) field. See *Exception priorities and pre-emption* on page B1-17 for more details. Only the pre-emption part of the priority is used by BASEPRI for masking.

[c] When used with the `MSR` instruction, it performs a conditional write. The BASEPRI value is only updated if the new priority is higher (lower number) than the current BASEPRI value.
Zero is a special value for BASEPRI (it means disabled). If BASEPRI is 0, it always accepts the new value. If the new value is 0, it will never accept it. This means BASEPRI_MAX can always enable BASEPRI but never disable it. PRIGROUP has no effect on the values compared or written. All register bits are compared and conditionally written.

[d] This register raises the ~~current~~ priority to -1 (the same as HardFault) when it is enabled set to 1. This can only be set by privileged code with a priority below -1 (not NMI or HardFault), and self-clears on return from any exception other than NMI. This is a 1-bit register.
The `CPS` instruction can also be used to update the FAULTMASK register.

[e] The control register is composed of the following bits:
[0] = Thread mode privilege: 0 means privileged, 1 means unprivileged (User). This bit resets to 0.
[1] = Current stack pointer: 0 is Main stack (MSP), 1 is alternate stack (PSP if Thread mode, RESERVED if Handler mode). This bit resets to 0.

Operation

```
if ConditionPassed() then
    case SYSm<7:3> of
        when '00000'
            if SYSm<2> == '0' then
                APSR<31:27> = R[n]<31:27>;
        when '00001'
            if CurrentModeIsPrivileged() then
                case SYSm<2:0> of
                    when '000'
                        MSP = R[n];
                    when '001'
                        PSP = R[n];
        when '00010'
            case SYSm<2:0> of
                when '000'
                    if CurrentModeIsPrivileged() then PRIMASK<0> = R[n]<0>;
                when '001'
                    if CurrentModeIsPrivileged() then BASEPRI<7:0> = R[n]<7:0>;
                when '010'
                    if CurrentModeIsPrivileged() &&
                    (R[n]<7:0> != '00000000') &&
                    (R[n]<7:0> < BASEPRI<7:0> || BASEPRI<7:0> == '00000000') then
                        BASEPRI<7:0> = R[n]<7:0>;
                when '011'
                    if CurrentModeIsPrivileged() &&
                    (ExecutionPriority > -1) then
                        FAULTMASK<0> = R[n]<0>;
                when `100`
                    if CurrentModeIsPrivileged() then
                        CONTROL<0> = R[n]<0>;
                        If CurrentMode == Mode_Thread then CONTROL<1> = R[n]<1>;
```

Exceptions

None.

Notes

Privilege Writes from unprivileged Thread mode to any stack pointer, the EPSR, the IPSR, the masks, or CONTROL, will be ignored. If privileged Thread mode software writes a ~~0~~ into CONTROL[0], the core will switch to unprivileged Thread mode (User) execution, and inhibit further writes to special-purpose registers.

An `ISB` instruction is required to ensure instruction fetch correctness following a Thread mode privileged => unprivileged transition.

IPSR The currently defined IPSR fields are not writable. Attempts to write them by Privileged code is write-ignored (has no effect).

EPSR The currently defined EPSR fields are not writable. Attempts to write them by Privileged code is write-ignored (has no effect).

Bit positions The PSR bits are positioned in each PSR according to their position in the larger xPSR composite. This is defined in *The special-purpose program status registers (xPSR)* on page B1-8.

Part C

Debug Architecture

Chapter C1
ARMv7-M Debug

This chapter covers all aspects of debug with respect to ARMv7-M. It is made up of the following sections:

This chapter is profile specific. ARMv7-M includes several debug features unique within the ARMv7 architecture to this profile.

C1.1 Introduction to debug

Debug support is a key element of the ARM architecture. ARMv7-M provides a range of debug approaches, both invasive and non-invasive techniques.

Invasive debug techniques are:

- the ability to halt the core, execute to breakpoints etc. (run-stop model)
- debug code using the DebugMonitor exception (less intrusive than halting the core).

Non-invasive debug techniques are:

- application trace by writing to the Instrumentation Trace Macrocell (ITM), a very low level of intrusion
- non-intrusive hardware supported trace and profiling,

Debug is normally accessed via the DAP (see *The Debug Access Port (DAP)* on page C1-4), which allows access to debug resources when the processor is running, halted, or held in reset. When a core is halted, the core is in Debug state.

The software-based and non-intrusive hardware debug features supported are as follows:

- High level trace and logging using the Instrumentation Trace Macrocell (ITM). This uses a fixed low-intrusion overhead (non-blocking register writes) which can be added to an RTOS, application or exception handler/ISR. The instructions can be retained in product code avoiding probe effects where necessary.
- Profiling a variety of system events including associated timing information. These include monitoring core clock counts associated with interrupt and sleep functions.
- PC sampling and event counts associated with load/store, instruction folding, and CPI statistics.
- Data tracing.

As well as the Debug Control Block (DCB) within the System Control Space (SCS), other debug related resources are allocated fixed 4kB address regions within the Private Peripheral Bus (PPB) region of the ARMv7-M system address map:

- Instrumentation Trace Macrocell (ITM) for profiling software.
- Debug Watchpoint and Trace (DWT) provides watchpoint support, program counter sampling for performance monitoring and embedded trace trigger control.
- Flash Patch and Breakpoint (FPB) block. This block can remap sections of ROM (Flash memory) to regions of RAM and set breakpoints on code in ROM. This feature can be used for debug and provision of code and/or data patches to applications where updates or corrections to product ROM(s) are required in the field.
- Embedded Trace Macrocell (ETM). This optional block provides instruction tracing.
- Trace Port Interface Unit (TPIU). This optional block provides the pin interface for the ITM, DWT and ETM (where applicable) trace features.

- ROM table. A table of entries providing a mechanism to identify the debug infrastructure supported by the implementation.

The address ranges for the ITM, DWT, FPB, DCB and trace support are listed in Table C1-1.

Note

Minimal systems might not include all the listed debug features, see *Debug support in ARMv7-M* on page C1-10.

Table C1-1 PPB debug related regions

Private Peripheral Bus (address range 0xE0000000 to 0xE00FFFFF)		
Group	**Address Offset Range(s)**	**Notes**
Instrumentation Trace Macrocell (ITM)	`0xE0000000-0xE0000FFF`	profiling and performance monitor support
Data Watchpoint and Trace (DWT)	`0xE0001000-0xE0001FFF`	includes control for trace support
Flash Patch and Breakpoint (FPB)	`0xE0002000-0xE0002FFF`	optional block
SCS: System Control Block (SCB)	`0xE000ED00-0xE000ED8F`	SCB: generic control features
SCS: Debug Control Block (DCB)	`0xE000EDF0-0xE000EEFF`	debug control and configuration
Trace Port Interface Unit (TPIU)	`0xE0040000-0xE0040FFF`	optional trace and/or serial wire viewer support (see notes).
Embedded Trace Macrocell (ETM)	`0xE0041000-0xE0041FFF`	optional instruction trace capability
ARMv7-M ROM table	`0xE00FF000-0xE00FFFFF`	DAP accessible for auto-configuration

Notes on Table C1-1:

- There is a requirement for writes to the ITM stimulus ports not to cause an exception when the ITM feature is disabled or not present to ensure the feature is transparent to application code, see *Theory of operation* on page C1-27.
- The SCB is described in *The System Control Block (SCB)* on page B3-10.
- In addition to the DWT and ITM, a TPIU is needed to export information off-chip for data trace, application trace, and profiling. The protocol used to communicate with an external software agent via the TPIU is described in Appendix E Debug ITM and DWT packet protocol.
- The TPIU can be a shared resource in a complex debug system, or omitted where visibility of ITM stimuli, or ETM and DWT trace event output is not required. Where the TPIU is a shared resource, it can reside within the PPB memory map and under local processor control, or be an external system resource, controlled from elsewhere.

C1.2 The Debug Access Port (DAP)

Debug access is through the Debug Access Port (DAP), an implementation of the *ARM Debug Interface v5 Architecture Specification*. A JTAG Debug Port (JTAG-DP) or Serial Wire Debug Port (SW-DP) can be used. The DAP specification includes details on how a system can be interrogated to determine what debug resources are available, and how to access any ARMv7-M device(s). A valid ARMv7-M system instantiation includes a ROM table of information as described in Table C1-3. The general format of a ROM table entry is described in Table C1-2.

A debugger can use a DAP interface to interrogate a system for memory access ports (MEM-APs). The BASE register in a memory access port provides the address of the ROM table (or a series of ROM tables within a ROM table hierarchy). The memory access port can then be used to fetch the ROM table entries. See *ARM Debug Interface v5 Architecture Specification* for more information.

Table C1-2 ROM table entry format

Bits	Name	Description
[31:12]	Address offset	Signed base address offset of the component relative to the ROM base address
[11:2]	Reserved	UNK/SBZP
[1]	Format	Reads-as-one when a valid table entry
[0]	Entry present	1: valid table entry 0: (and bits [31:1] not equal to zero), ignore the table entry[a]

a. 0x00000002 is the recommended null entry for ARMv7-M where a null entry is required before an end of table marker.

For ARMv7-M all address offsets are negative. The entry 0x00000000 indicates the end of table marker.

Table C1-3 ARMv7-M DAP accessible ROM table

Offset	Value	Name	Description
`0x000`	`0xFFF0F003`	ROMSCS	Points to the SCS at 0xE000E000.
`0x004`	`0xFFF02002` or `0xFFF02003`	ROMDWT	Points to the Data Watchpoint and Trace block at 0xE0001000. Bit [0] is set if a DWT is fitted.
`0x008`	`0xFFF03002` or `0xFFF03003`	ROMFPB	Points to the Flash Patch and Breakpoint block at 0xE0002000. Bit [0] is set if an FPB is fitted.
`0x00C`	`0xFFF01002` or `0xFFF01003`	ROMITM[a]	Points to the Instrumentation Trace block at 0xE0000000. Bit [0] is set if an ITM is fitted.
`0x010`	`0xFFF41002` or `0xFFF41003`	ROMTPIU[b]	Points to the Trace Port Interface Unit. Bit [0] is set if a TPIU is fitted and accessible to the processor on its Private Peripheral Bus (PPB).

Table C1-3 ARMv7-M DAP accessible ROM table (continued)

Offset	Value	Name	Description
`0x014`	`0xFFF42002` or `0xFFF42003`	ROMETM[b]	Points to the Embedded Trace Macrocell block. Bit [0] is set if an ETM is fitted and accessible to the processor on its PPB.
`0x018`	`0x00000000`	end	End-of-table marker. It is IMPLEMENTATION DEFINED whether the table is extended with pointers to other system debug resources. The table entries always terminate with a null entry.
`0x020` to `0xFC8`		Not Used	RAZ
`0xFCC`	`0x00000001`	SYSTEM ACCESS[c]	Bit [0] set indicates that resources other than those listed in the ROM table are accessible wihin the same 32-bit address space via the DAP.
`0xFD0`	IMP DEF	PID4	CIDx values are fully defined for the ROM table, and are CoreSight compliant. PIDx values should be CoreSight compliant or RAZ. See Appendix B *ARMv7-M infrastructure IDs* for more information. CoreSight: ARM's system debug architecture
`0xFD4`	0	PID5	
`0xFD8`	0	PID6	
`0xFDC`	0	PID7	
`0xFE0`	IMP DEF	PID0	
`0xFE4`	IMP DEF	PID1	
`0xFE8`	IMP DEF	PID2	
`0xFEC`	IMP DEF	PID3	
`0xFF0`	`0x0000000D`	CID0	
`0xFF4`	`0x00000010`	CID1	
`0xFF8`	`0x00000005`	CID2	
`0xFFC`	`0x000000B1`	CID3	

[a] Accesses cannot cause a non-existent memory exception.

[b] It is IMPLEMENTATION DEFINED whether a shared resource is managed by the local processor or a different resource.

[c] This location was formerly known as MEMTYPE.

The basic sequence of events to access and enable ARMv7-M debug using a DAP is as follows:

- Enable the power-up bits for the debug logic in the DAP Debug Port control register.
- Ensure the appropriate DAP Memory Access Port control register is enabled for word accesses (this should be the default in a uniprocessor system).

- If halting debug is required:
  - set the C_DEBUGEN bit in the Debug Halting Control and Status Register (DHCSR) – see *Debug Halting Control and Status Register (DHCSR)* on page C1-20.

  If the target is to be halted immediately:
  - set the C_HALT bit in the same register
  - read back the S_HALT bit in the DHCSR to ensure the target is halted in Debug state.

  Otherwise, if monitor debug is required:
  - enable DebugMonitor exceptions in the DEMCR.

    Note

    C_DEBUGEN must be clear if DebugMonitor exceptions are to occur. If C_DEBUGEN is set, halting debug behavior overrides DebugMonitor exceptions.

- If using the watchpoint and trace features, set the TRCENA bit in the Debug Exception and Monitor Control Register (DEMCR) – see *Debug Exception and Monitor Control Register (DEMCR)* on page C1-24.

See the *ARM Debug Interface v5 Architecture Specification* for more information on the DAP.

Warning

System control and configuration fields (in particular registers in the SCB) can be changed via the DAP while software is executing. For example, resources designed for dynamic updates can be modified. This can have undesirable side-effects if both the application and debugger are updating the same or related resources. The consequences of updating a running system via a DAP in this manner have no guarantees, and can be worse than UNPREDICTABLE with respect to system behavior.

In general, MPU or FPB address remapping changes should not be performed by a debugger while software is running to avoid associated context problems.

C1.2.1 General rules applying to debug register access

The Private Peripheral Bus (PPB), address range `0xE0000000` to `0xE0100000`, supports the following general rules:

- The region is defined as Strongly Ordered memory – see *Strongly-ordered memory* on page A3-25 and *Memory access restrictions* on page A3-26.
- Registers are always accessed little endian regardless of the endian state of the processor.
- Debug registers can only be accessed as a word access. Byte and halfword accesses are UNPREDICTABLE.
- The term *set* means assigning the value to 1 and the term clear(ed) means assigning the value to 0. Where the term applies to multiple bits, all bits assume the assigned value.

- The term *disable* means assigning the bit value to 0 and the term enable means assigning the bit value to 1.
- A reserved register or bit field has the value UNK/SBZP.

Unprivileged (User) access to the PPB causes BusFault errors unless otherwise stated. Notable exceptions are:

- Unprivileged accesses can be enabled to the Software Trigger Interrupt Register in the System Control Space by programming a control bit in the Configuration Control Register.
- ~~For debug related resources (DWT, ITM, FPB, ETM and TPIU blocks), user access reads return UNKNOWN and writes are ignored unless stated otherwise.~~

C1.3 Overview of the ARMv7-M debug features

ARMv7-M defines a debug model specifically designed for the profile. The ARMv7-M debug model has control and configuration integrated into the memory map. The Debug Access Port defined in the *ARM Debug Interface v5 Architecture Specification* provides the interface to a host debugger. Debug resources within ARMv7-M are as listed in Table C1-1 on page C1-3.

ARMv7-M supports the following debug related features:

- A Local Reset, see *Overview of the exceptions supported* on page B1-14. This resets the core and supports debug of reset events.
- Core halt. Control register support to halt the processor. This can occur asynchronously by assertion of an external signal, execution of a `BKPT` instruction, or from a debug event (by example configured to occur on reset, or on exit from or entry to an ISR).
- Step, with or without interrupt masking.
- Run, with or without interrupt masking.
- Register access. The DCB supports debug requests, including reading and writing core registers when halted.
- Access to exception-related information through the SCS resources. Examples are the currently executing exception (if any), the active list, the pended list, and the highest priority pending exception.
- Software breakpoints. The `BKPT` instruction is supported.
- Hardware breakpoints, hardware watchpoints, and support for remapping of code memory locations.
- Access to all memory through the DAP.
- Support of profiling. Support for PC sampling is provided.
- Support of instruction tracing and the ability to add other system debug features such as a bus monitor or cross-trigger facility. ETM instruction trace requires a multiwire Trace Port Interface Unit (TPIU).
- Application and data trace that can be supported through either a low pin-count Serial Wire Viewer (SWV) or a multiwire TPIU.

Note

CoreSight is the name given to ARM's system debug architecture, incorporating a range of debug control, capture and system interface blocks. ARMv7-M does not require CoreSight compliance. The register definitions and address space allocations for the DWT, ITM, TPIU and FPB blocks in this specification are compatible. ARMv7-M allows these blocks to add support for CoreSight topology detection and operation as appropriate by extending them with CoreSight ID and management registers.

C1.3.1 Debug authentication

ARMv7 supports two generic signals for debug enable and to control invasive versus non-invasive debug as described in Table C1-4.

Table C1-4 ARMv7 debug authentication signals

DBGEN	NIDEN	Invasive debug permitted	Non-invasive debug permitted
LOW	LOW	No	No
LOW	HIGH	No	Yes
HIGH	X	Yes	Yes

For ARMv7-M, the provision of **DBGEN** and **NIDEN** as actual signals is IMPLEMENTATION DEFINED. It is acceptable for **DBGEN** to be considered permanently enabled (**DBGEN** = HIGH), with control deferred to other enable bits within the profile specific debug architecture.

Note

ARMv7-M does not support the **SPIDEN** and **SPNIDEN** signals. These signals form part of the secure debug authentication scheme as used by ARMv6K with the Security Extensions and the ARMv7-A profile.

C1.3.2 External debug request

The **EDBGRQ** input is asserted by an external agent to signal an external debug request. An external debug request can cause a debug event and entry to Debug state as described in *Debug event behavior* on page C1-14. The debug event is reported in the DFSR.EXTERNAL status bit, see *Debug Fault Status Register (DFSR)* on page C1-19.

When the processor is in Debug state, the **HALTED** output signal is asserted. **HALTED** reflects the DHCSR.S_HALT bit, see *Debug Halting Control and Status Register (DHCSR)* on page C1-20. The signal can be used as a debug acknowledge for **EDBGRQ**.

EDBGRQ and **HALTED** assert HIGH. **EDBGRQ** is ignored when the processor is in Debug state.

C1.3.3 External restart request

It is IMPLEMENTATION DEFINED whether multiprocessing support is provided in the ARMv7-M Debug Extension. An implementation with multiprocessing debug support is required to provide the ability to perform a linked restart of multiple processors. Two signals are required to support the multiprocessing restart mechanism:

- a **DBGRESTART** input
- a **DBGRESTARTED** output.

DBGRESTART and DBGRESTARTED

DBGRESTART and **DBGRESTARTED** form a four-phase handshake, as shown in Figure C1-1.

Asserting **DBGRESTART** HIGH causes the core to exit from Debug state. Once **DBGRESTART** is asserted, it must be held HIGH until **DBGRESTARTED** is deasserted. **DBGRESTART** is ignored unless **HALTED** and **DBGRESTARTED** are asserted.

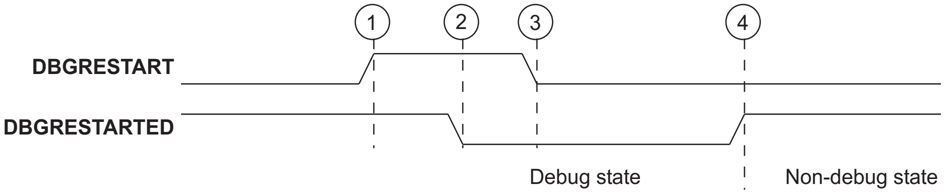


Figure C1-1 DBGRESTART / DBGRESTARTED handshake

Figure C1-1 is diagrammatic only, and no timings are implied. The numbers in Figure C1-1 have the following meanings:

1. If **DBGRESTARTED** is asserted HIGH the peripheral asserts **DBGRESTART** HIGH and waits for **DBGRESTARTED** to go LOW
2. The processor drives **DBGRESTARTED** LOW to deassert the signal and waits for **DBGRESTART** to go LOW
3. The peripheral drives **DBGRESTART** LOW to deassert the signal. This is the event that indicates to the processor that it can start the Debug → Non-debug state transition phase.
4. The processor leaves Debug state and asserts **DBGRESTARTED** HIGH.

In the process of leaving Debug state the processor clears the **HALTED** signal to LOW. It is IMPLEMENTATION DEFINED when this change occurs relative to the 1 → 0 change in **DBGRESTART** and the 0 → 1 change in **DBGRESTARTED**.

C1.3.4 Debug support in ARMv7-M

ARMv7-M supports a comprehensive set of debug features. The following bit fields can be used to determine the level of debug support present in a design:

- if ROMDWT[0] is zero there is no DWT support. Otherwise, if DEMCR.TRCENA == 1 and if:
  - DWT_CTRL.NOTRCPKT == ’1’, there is no DWT trace sampling or exception tracing support
  - DWT_CTRL.NOEXTTRIG == ’1’, there is no **CMPMATCH[N]** support
  - DWT_CTRL.NOCYCCNT == ’1’, there is no cycle counter support
  - DWT_CTRL.NOPRFCNT == ’1’, there is no profiling counter support.
- if ROMITM[0] is zero there is no ITM support

- if ROMFPB[0] is zero there is no FPB support
- if ROMETM[0] is zero there is no ETM support
- if neither DWT nor ITM is supported, DEMCR.TRCENA is RAZ/WI
- if FP_REMAP[29] is zero the FPB only supports breakpoint functionality.

Note

The number of comparators supported in the DWT or FPB can be determined from bit fields in the DWT_CTRL and FP_CTRL registers respectively.

Recommended levels of debug

There are three recommended levels of debug provision in ARMv7-M:

- a minimum level that only supports the DebugMonitor exception
- a basic level that requires a DAP and adds some halting debug support
- a comprehensive level that includes the above with fully-featured ITM, DWT and FPB support.

The minimum level of debug in ARMv7-M only supports core access (no DAP) and the DebugMonitor exception with:

- the `BKPT` instruction

  Note

  When the DebugMonitor exception is disabled, this escalates to a HardFault exception.

- monitor stepping
- monitor entry from **EDBGRQ**.

ARM defines the following configuration of features as a basic level of support:

- support of a DAP and halting debug
- no ITM support - ROMITM[0] == ’0’, see *ARMv7-M DAP accessible ROM table* on page C1-4

  Note

  There is a requirement for writes to the ITM stimulus ports not to cause an exception when the ITM feature is disabled or not present to ensure the feature is transparent to application code, see *Theory of operation* on page C1-27.

- 2 breakpoints in the FPB (no remapping support)
- 1 watchpoint in the DWT (no trace sampling or external match signal (**CMPMATCH[N]**) generation)
- Debug monitor support of the minimum level debug features along with the listed FPB and DWT events.

For compliance with the comprehensive level of support:

- the DebugMonitor exception and halting debug are supported
- ROMITM[0] != ’0’
  - at least 8 Stimulus Port registers.
- ROMDWT[0] != ’0’
  - at least 1 watchpoint is supported
  - DWT_CTRL.NOTRCPKT == ’0’
  - DWT_CTRL.NOCYCCNT == ’0’
  - DWT_CTRL.NOPRFCNT == ’0’
  - DWT_CTRL.NOEXTTRIG is IMPLEMENTATION DEFINED.

    CMPMATCH[N] support is required when ROMETM[0] == ’1’.
- ROMFPB[0] != ’0’
  - at least 2 breakpoints are supported
  - FP_REMAP[29] != ’0’.

C1.4 Debug and reset

ARMv7-M defines two levels of reset as stated in *Overview of the exceptions supported* on page B1-14:

- a Power-ON Reset (POR)
- a Local Reset.

Software can initiate a system reset as described in *Reset management* on page B1-47. The reset vector catch control bit (VC_CORERESET) can be used to generate a debug event when the core comes out of reset. A debug event causes the core to halt (enter Debug state) when halting debug is enabled.

The following bit fields are reset by a POR but not by a Local Reset:

- fault flags in the DFSR, see *Debug Fault Status Register (DFSR)* on page C1-19.
- debug control in the DHCSR, see the notes associated with *Debug Halting Control and Status Register (DHCSR)* on page C1-20.
- DEMCR.TRCENA and the vector catch (VC*) configuration bits, see *Debug Exception and Monitor Control Register (DEMCR)* on page C1-24.

Note

ARMv7-M does not provide a means to:

- debug a Power-On Reset
- differentiate Power-On Reset from a Local Reset.

The relationship with the debug logic reset and power control signals described in the DAP recommended external interface (see *ARM Debug Interface v5 Architecture Specification*) is IMPLEMENTATION DEFINED.

C1.5 Debug event behavior

An event triggered for debug reasons is known as a debug event. A debug event will cause one of the following to occur:

- Entry to Debug state. If halting debug is enabled (C_DEBUGEN in the DHCSR, Table C1-9 on page C1-20, is set), captured events will halt the processor in Debug state. See Table B1-9 on page B1-40 for a comprehensive application level fault table.
- A DebugMonitor exception. If halting debug is disabled (C_DEBUGEN is cleared) and the debug monitor is enabled (MON_EN in the DEMCR, Table C1-12 on page C1-24, is set), a debug event will cause a DebugMonitor exception when the group priority of DebugMonitor is higher than the current active group priority.

  If the DebugMonitor group priority is less than or equal to the current active group priority~~, a BKPT instruction will escalate to a HardFault and other debug events (watchpoints and external debug requests) are ignored~~.

  Note

  Software can ~~put the DebugMonitor exception into the Pending state under this condition, and when the DebugMonitor exception is disabled~~.

- A HardFault exception. If both halting debug and the monitor are disabled, a BKPT instruction will escalate to a HardFault and other debug events (watchpoints and external debug requests) are ignored.

  Note

  A BKPT instruction that causes a HardFault or lockup is considered as unrecoverable.

The Debug Fault Status Register (Table C1-8 on page C1-19) contains status bits for each captured debug event. The bits are write-one-to-clear. These bits are set when a debug event causes the processor to halt or generate an exception. It is IMPLEMENTATION DEFINED whether the bits are updated when an event is ignored.

A summary of halting and debug monitor support is provided in Table C1-5.

Table C1-5 Debug related faults

Fault Cause	Exception support (Halt and DebugMonitor)	DFSR Bit Name	Notes
Internal halt request	Yes	HALTED	Step command, core halt request, etc.
Breakpoint	Yes	BKPT	Breakpoint from BKPT instruction or match in FPB
Watchpoint[a]	Yes	DWTTRAP	Watchpoint match in DWT
Vector catch	Halt only	VCATCH	DEMCR.VC_xxx bit(s) set
External	Yes	EXTERNAL	EDBGRQ line asserted

[a] Includes a PC match watchpoint.

For a description of the vector catch feature, see *Vector catch support* on page C1-26.

If DHCSR.C_DEBUGEN is clear and a breakpoint occurs in an NMI or HardFault exception handler, the system locks up with an unrecoverable error. Handling of unrecoverable exceptions in general is described in *Unrecoverable exception cases* on page B1-44. The breakpoint can be due to a BKPT instruction or generated by the FPB, see *Flash Patch and Breakpoint (FPB) support* on page C1-61.

C1.5.1 Debug stepping

ARMv7-M supports debug stepping in both halting debug and monitor debug. Stepping from Debug state is supported by writing to the C_STEP and C_HALT control bits in the Debug Halt Control and Status Register (see *Debug Halting Control and Status Register (DHCSR)* on page C1-20 for the control bit definitions).

When C_STEP is set, and C_HALT is cleared in the same or a subsequent register write, the system:

1. exits Debug state
2. performs one of the following:
   - The *next instruction* is executed (stepped).
   - An exception entry sequence occurs that stacks the next instruction context. The processor halts on the first instruction of the exception handler entered according to the exception priority and late-arrival rules.

- The *next instruction* is executed (stepped) and the exception model causes a change from the expected program flow:
  - — An exception entry sequence occurs according to the exception priority and late-arrival rules. The processor halts ready to execute the first instruction of the exception handler taken.
  - — If the executed instruction is an exception return instruction, tail-chaining can cause entry to a new exception handler. The processor halts ready to execute the first instruction of the exception handler taken.

Note

The exception entry behavior is not recursive. Only a single `PushStack()` update can occur within a step sequence.

3. returns to Debug state.

The debugger can optionally set the C_MASKINTS bit in the DHCSR to inhibit (mask) PendSV, Systick and external configurable interrupts from occurring. Where C_MASKINTS is set, permitted exception handlers which activate will execute along with the stepped instruction. See Table C1-6 for a summary of stepping control.

Table C1-6 Debug stepping control using the DHCSR

DHCSR writes[a]			
C_HALT	**C_STEP**	**C_MASKINTS**	**Action**
0	0	0	Exit Debug state and start instruction execution Exceptions activate according to the exception configuration rules.
0	0	1	Exit Debug state and start instruction execution. PendSV, SysTick and external configurable interrupts are disabled, otherwise exceptions activate according to standard configuration rules.
0	1	0	Exit Debug state, step an instruction and halt. Exceptions activate according to the exception configuration rules.

Table C1-6 Debug stepping control using the DHCSR (continued)

DHCSR writes[a]			
C_HALT	C_STEP	C_MASKINTS	Action
0	1	1	Exit Debug state, step an instruction and halt. PendSV, SysTick and external configurable interrupts are disabled, otherwise exceptions activate according to standard configuration rules.
1	x	x	remain in Debug state

[a] assumes C_DEBUGEN == 1 and S_HALT == 1 when the write occurs (the system is halted).

~~Modifying C_STEP or C_MASKINTS while the system is running with halting debug support enabled (C_DEBUGEN == 1, S_HALT == 0) is UNPREDICTABLE.~~

The values of C_HALT, C_STEP and C_MASKINTS are ignored by hardware and UNKNOWN to software when C_DEBUGEN == 0.

Note

If C_HALT is cleared in Debug state, a subsequent read of S_HALT == ’1’ means Debug state has been re-entered due to detection of a new debug event.

Debug monitor stepping

Stepping by a debug monitor is supported by the MON_STEP control bit in the Debug Exception and Monitor Control Register (see *Debug Exception and Monitor Control Register (DEMCR)* on page C1-24). ~~When MON_STEP is set (with C_DEBUGEN clear), the step request is a Pending request that will activate on return from the DebugMonitor handler to the code being debugged (the debug target code).~~

~~**Note**~~

~~Tail-chaining can result in other exception handlers being executed before the monitor step request is activated.~~

~~Once the step request activates, it performs one of the following in the step execution phase:~~

- ~~The *next instruction* is executed (stepped).~~
- ~~An exception entry sequence occurs that stacks the next instruction context. The processor halts on the first instruction of the exception handler entered according to the exception priority and late-arrival rules.~~

- ~~The *next instruction* is executed (stepped) and the exception model causes a change from the expected program flow:~~
  - ~~An exception entry sequence occurs according to the exception priority and late-arrival rules. The processor halts ready to execute the first instruction of the exception handler taken.~~
  - ~~If the executed instruction is an exception return instruction, tail-chaining can cause entry to a new exception handler. The processor halts ready to execute the first instruction of the exception handler taken.~~

~~Note~~

~~Only a single `PushStack()` update can occur due to a non-recursive exception entry sequence within the step execution phase.~~

~~After the step execution phase, the DebugMonitor exception will be taken with the DFSR.HALTED bit set, see *Debug Fault Status Register (DFSR)* on page C1-19.~~

C1.6 Debug register support in the SCS

The debug provision in the System Control Block consists of two handler-related flag bits (ISRPREEMPT and ISRPENDING) in the Interrupt Control and State Register (*Interrupt Control and State Register – (0xE000ED04)* on page B3-12), and the Debug Fault Status Register (DFSR).

Additional debug registers are architected in the Debug Control Block as summarized in Table C1-7.

Table C1-7 Debug register region of the SCS

Address	R/W	Name	Function
`0xE000EDF0`	R/W	DHCSR	Debug Halting Control and Status Register
`0xE000EDF4`	WO	DCRSR	Debug Core Register Selector Register
`0xE000EDF8`	R/W	DCRDR	Debug Core Register Data Register
`0xE000EDFC`	R/W	DEMCR	Debug Exception and Monitor Control Register
… to `0xE000EEFF`	…		Reserved for debug extensions

C1.6.1 Debug Fault Status Register (DFSR)

The Debug Fault Status Register as defined in Table C1-8 provides the top level reason why a debug event has occurred.

Table C1-8 Debug Fault Status Register (0xE000ED30)

Bits[a]	R/W	Name	Function
[31:5]			Reserved
[4]	R/W1C[b]	EXTERNAL	An asynchronous exception generated due to the assertion of **EDBGRQ**.
[3]	R/W1C	VCATCH	Vector catch triggered. Corresponding FSR will contain the primary cause of the exception.
[2]	R/W1C	DWTTRAP	Data Watchpoint and Trace trap. Indicates that the core halted due to at least one DWT trap event.
[1]	R/W1C	BKPT	`BKPT` instruction executed or breakpoint match in FPB.
[0]	R/W1C	HALTED	~~Halt request, including step debug command. Stopped on next instruction.~~

a. bits [4:0] are cleared on a power-up reset. They are not cleared by a software initiated (local) reset.

b. R/W1C: Read/Write-one-to-clear.

C1.6.2 Debug Halting Control and Status Register (DHCSR)

The Debug Halting Control and Status Register (DHCSR) controls halting debug.

Table C1-9 Debug Halting Control and Status Register – (0xE000EDF0)

Bits	R/W	Name	Function
[31:16]	W	DBGKEY	Debug Key. The value `0xA05F` must be written to enable write accesses to bits [15:0], otherwise the write access will be ignored. Read behavior of bits [31:16] is as listed below.
[31:26]	R	-	Reserved
[25]	R	S_RESET_ST	Core reset since the last time this bit was read. This is a sticky bit, which clears on read.
[24]	R	S_RETIRE_ST	~~Instruction has completed (retired) since last read. This is a sticky bit, which clears on read.~~ This bit can be used to determine if the core is stalled on a load/store or fetch.
[23:20]	-	-	Reserved
[19]	R	S_LOCKUP	Core is locked up due to an unrecoverable exception. See *Unrecoverable exception cases* on page B1-44 for details. This bit can only be read as set by a remote debugger (via the DAP) while the core is running and locked up. The bit is cleared on entry to Debug state when the core halts.
[18]	R	S_SLEEP	Core is sleeping. Must set the C_HALT bit to gain control, or wait for an interrupt (`WFI` instruction response) to wake-up the system.
[17]	R	S_HALT	Core is in Debug state.
[16]	R	S_REGRDY	A handshake flag. The bit is cleared to ’0’ on a write to the Debug Core Register Selector Register and is set to ’1’ when the transfer to/from the Debug Core Register Data Register is complete. ~~Otherwise~~ the bit is UNKNOWN.
[15:6]	-	-	Reserved
[5]	R/W	C_SNAPSTALL	~~If the core is stalled on a load/store operation (see S_RETIRE_ST), setting this bit will break the stall and force the instruction to complete. This bit can only be set if C_DEBUGEN and C_HALT are set, and S_RETIRE_ST is clear. The bus state is UNPREDICTABLE when C_SNAPSTALL is set.~~
[4]	-	-	Reserved

Table C1-9 Debug Halting Control and Status Register – (0xE000EDF0) (continued)

Bits	R/W	Name	Function
[3]	R/W	C_MASKINTS	Mask PendSV, SysTick and external configurable interrupts when debug is enabled. The bit does not affect NMI. When C_DEBUGEN==0, this bit is UNKNOWN.
[2]	R/W	C_STEP	Step the core. When C_DEBUGEN==0, this bit is UNKNOWN.
[1]	R/W	C_HALT	Halt the core. When in Debug state, a write of this bit affects the core according to Table C1-6 on page C1-16. The bit reads as UNKNOWN.
[0]	R/W	C_DEBUGEN	Enable halting debug. C_MASKINTS must be written with 0 when this bit is asserted (written with 1 when previously 0). This bit can only be set from the DAP, it cannot be set under software control.

Notes on Table C1-9 on page C1-20:

- S_RESET_ST is set on every reset (power-on and local resets)
- ~~S_RETIRE_ST, S_LOCKUP, S_SLEEP and S_HALT clear on reset. If C_HALT and C_DEBUGEN are asserted on reset, S_HALT will be set, and the core will enter Debug state immediately after the reset sequence.~~
- ~~C_SNAPSTALL, C_MASKINTS, C_STEP, and C_DEBUGEN are cleared on a power-on reset only~~
- Modifying C_STEP or C_MASKINTS while the system is running with halting debug support enabled (C_DEBUGEN == 1, S_HALT == 0) is UNPREDICTABLE.
- For more information on the use of C_HALT, C_STEP and C_MASKINTS, see *Debug stepping* on page C1-15.

C1.6.3 Debug Core Register Selector Register (DCRSR)

The DCRSR write-only register generates a handshake to the core to transfer the selected register to/from the DCRDR. The DHCSR S_REGRDY bit is cleared when the DCRSR is written, and remains clear until the core transaction completes. This register is only accessible from Debug state.

Table C1-10 Debug Core Register Selector Register – (0xE000EDF4)

Bits	R/W	Name	Function
[31:17]	-	-	Reserved
[16]	WO	REGWnR	Write = 1, Read = 0
[15:5]	-	-	Reserved
[4:0]	WO	REGSEL	00000: R0 00001: R1 … 01100: R12 01101: the current SP 01110: LR 01111: DebugReturnAddress~~()~~ 10000: xPSR / Flags, Execution Number, and state information 10001: MSP (Main SP) 10010: PSP (Process SP) 10100: CONTROL bits [31:24], FAULTMASK bits [23:16], BASEPRI bits [15:8], and PRIMASK bits [7:0] All unused values reserved

Notes on Table C1-10:

xPSR etc For information on the xPSR see *The special-purpose program status registers (xPSR)* on page B1-8.

Note

A debugger must preserve the Exception Number (IPSR bits) in Debug state, otherwise the behavior is UNPREDICTABLE.

DebugReturnAddress~~()~~

The address of the next instruction to be executed on exit from Debug state.

The address reflects the point in the execution stream where the debug event was invoked. For a hardware or a software breakpoint, the address is the breakpointed instruction address.

For all other debug events, including PC match watchpoints, the address is that of an instruction which in a simple execution model is one which executes after the instruction which caused the event, but which itself has not executed. All instructions prior to this instruction in the model have executed.

Note

Bit [0] of DebugReturnAddress() is RAZ/WI. Writing bit [0] does not affect the EPSR.T-bit, which is accessed independently through the xPSR register selection.

C1.6.4 Debug Core Register Data Register (DCRDR)

The DCRDR is used with the DCRSR to provide access to the general-purpose and special-purpose registers in the core. The DCRDR is read or written (depending on REGWnR) to/from the selected register (defined by REGSEL) on a write to the DCRSR.

Note

1. The xPSR registers are fully accessible using this access method, unlike the `MSR`/`MRS` instructions where some bits RAZ and ignored on writes. Illegal values can cause faults to occur.
2. The EPSR.IT bits can be written. IT bits must only be written with a value consistent with the instruction to be executed on return from Debug state, otherwise instruction execution will be UNPREDICTABLE. See *ITSTATE* on page A6-10 for information on the correct bit patterns within an `IT` instruction block. The IT bits must be zero on exit from Debug state if the next instruction to execute is outside an `IT` block.
3. The EPSR.ICI bits can be written and used with an `LDM` or `STM` instruction. Clearing the ICI bits will cause the underlying `LDM` or `STM` instruction to restart instead of continue.
4. FAULTMASK cannot be set by an `MSR` instruction when the execution priority is -1 or higher (NMI, HardFault and exception handlers with FAULTMASK already set). See *MSR (register)* on page B4-8. This restriction does not apply to the DCRSR/DCRDR debug mechanism.

The DCRDR can be used on its own as a message passing resource between a debugger and a debug agent running on the core.

Table C1-11 Debug Core Register Data Register – (0xE000EDF8)

Bits	R/W	Name	Function
[31:0]	R/W	DBGTMP	Data temporary cache, for reading and writing registers. This register is UNKNOWN on reset or while DHCSR.S_REGRDY == ‘0’ during execution of a DCRSR based transaction that updates the register.

C1.6.5 Debug Exception and Monitor Control Register (DEMCR)

The DEMCR is used to manage exception behavior under debug. The register is used for vector catching and DebugMonitor handling.

Bits [23:16] are used for DebugMonitor exception control, and bits [15:0] are associated with Debug state (halting debug). All register bits are reset by a power-on reset. Only the DebugMonitor related bits (MONxxx) are cleared on a Local Reset, see *Reset behavior* on page B1-20 for the definition of Local Reset.

Table C1-12 provides the DEMCR details. For a complete list of faults and their assignment to vector catch enable bits, see *Fault behavior* on page B1-39.

Table C1-12 Debug Exception and Monitor Control Register – (0xE000EDFC)

Bits	R/W	Name	Function
[31:25]	-	-	Reserved
[24]	R/W	TRCENA	Global enable for all features configured and controlled by the DWT and ITM blocks. The bit can be used to gate the ETM and TPIU blocks too. Clearing the bit does not in itself guarantee stopping all events. The feature enables in the DWT and ITM blocks must be cleared before clearing TRCENA to ensure everything is stopped. This bit is cleared on a Power-up reset only. TRCENA is not cleared by a software initiated reset. When TRCENA == ’0’: • DWT registers return UNKNOWN values on reads. It is IMPLEMENTATION DEFINED whether writes to the DWT unit are ignored. • ITM registers return UNKNOWN values on reads. It is IMPLEMENTATION DEFINED whether writes to the ITM unit are ignored. If the ITM and DWT blocks are not implemented, TRCENA is RAZ/WI.
[23:20]	-	-	Reserved
[19]	R/W	MON_REQ	A DebugMonitor semaphore bit. The meaning is IMPLEMENTATION DEFINED by the monitor software. MON_REQ is cleared on a Local Reset.
[18]	R/W	MON_STEP	When MON_EN is set, this bit is used to step the core. When MON_EN is clear, this feature is disabled. This is the debug monitor equivalent of C_STEP in Debug state. MON_STEP is cleared on a Local Reset.

Table C1-12 Debug Exception and Monitor Control Register – (0xE000EDFC) (continued)

Bits	R/W	Name	Function
[17]	R/W	MON_PEND	Pend the DebugMonitor exception to activate when priority allows. This can be use to wake up the monitor via the DAP. MON_PEND is cleared on a Local Reset.
[16]	R/W	MON_EN	Enable the DebugMonitor exception. See *Exception model* on page B1-14 for the exception model details. When disabled BKPT instructions are escalated to HardFault, FPB generated breakpoints are IMPLEMENTATION DEFINED[a], and all other DebugMonitor exceptions are ignored. C_DEBUGEN (halting debug) in the DHCSR overrides this bit. MON_EN is cleared on a Local Reset.
[15:11]	-	-	Reserved
[10]	R/W	VC_HARDERR[b]	Debug trap on a HardFault exception. Ignored when C_DEBUGEN is clear.
[9]	R/W	VC_INTERR[b]	Debug trap on a fault occurring during an exception entry or return sequence. Ignored when C_DEBUGEN is clear.
[8]	R/W	VC_BUSERR[b]	Debug trap on a BusFault exception. Ignored when C_DEBUGEN is clear.
[7]	R/W	VC_STATERR[b]	Debug trap on a UsageFault exception due to a state information error (for example an UNDEFINED instruction). Ignored when C_DEBUGEN is clear.
[6]	R/W	VC_CHKERR[b]	Debug trap on a UsageFault exception due to a checking error (for example an alignment check error). Ignored when C_DEBUGEN is clear.
[5]	R/W	VC_NOCPERR[b]	Debug trap on a UsageFault access to a Coprocessor. Ignored when C_DEBUGEN is clear.
[4]	R/W	VC_MMERR[b]	Debug trap on a MemManage exception. Ignored when C_DEBUGEN is clear.
[3:1]	-	-	Reserved
[0]	R/W	VC_CORERESET[b]	Reset Vector Catch. Halt a running system when a Local Reset occurs. Ignored when C_DEBUGEN is clear.

a. It is IMPLEMENTATION DEFINED whether an FPB generated breakpoint takes a HardFault or is ignored. An FPB generated breakpoint can only be ignored where the breakpointed instruction exhibits its normal architectural behavior.

b. The vector catch (VC prefixed) bits are cleared on a power-up reset. They are not altered by a software initiated reset.

Vector catch support

Vector catch support is the mechanism used to generate a debug event and enter Debug state when a particular exception occurs. Vector catching is only supported by halting debug.

If C_DEBUGEN in the DHCSR is set, at least one VC* enable bit is set in the DEMCR, and the associated exception activates, then a debug event occurs. This causes Debug state to be entered (execution halted) on the first instruction of the exception handler.

Note

Fault status bits are set on exception entry and are available to the debugger to help determine the source of the error (see *Configurable Fault Status Registers (UserFault, BusFault, and MemManage)* on page B3-18, *HardFault Status register (HFSR)* on page B3-21 and *Debug Fault Status Register (DFSR)* on page C1-19).

A vector catch guarantees to enter Debug state without executing any additional instructions. However, saved context might include information on a lockup situation or a higher priority pending exception, for example a pending NMI exception detected on reset.

Late arrival and derived exceptions can occur, postponing when the processor will halt. See *Late arriving exceptions* on page B1-33 and *Derived exceptions* on page B1-34 for details.

C1.7 Instrumentation Trace Macrocell (ITM) support

The Instrumentation Trace Macrocell (ITM) provides a memory-mapped register interface to allow applications to write logging/event words to the optional external Trace Port Interface Unit (TPIU). The ITM also supports control and generation of timestamp information packets.

The event words and timestamp information are formed into packets and multiplexed with hardware event packets from the Data Watchpoint and Trace (DWT) block according to the packet protocol described in Appendix E *Debug ITM and DWT packet protocol*.

C1.7.1 Theory of operation

The ITM consists of:

- stimulus (Stimulus Port) registers
- stimulus enable (Trace Enable) registers
- a stimulus access (Trace Privilege) register
- a general control (Trace Control) register.

The number of Stimulus Port registers is an IMPLEMENTATION DEFINED multiple of eight. Writing all 1s to the Trace Privilege Register then reading how many bits are set can be used to determine the number of Stimulus Ports supported.

The Trace Privilege Register defines whether the associated Stimulus Ports (in groups of 8) and their corresponding Trace Enable Register bits can be written by an unprivileged (User) access. User code can always read the Stimulus Ports.

Stimulus Port registers are 32-bit registers that support word-aligned (address[1:0] == 0b00) byte (bits [7:0]), halfword (bits [15:0]), or word accesses. Non-word-aligned accesses are UNPREDICTABLE. There is a global enable, ITMENA, in the control register and additional mask bits, which enable the Stimulus Port registers individually, in the Trace Enable Register.

When an enabled Stimulus Port is written to, the identity of the port, the size of the write access, and the data written are copied into a FIFO for emission to a trace sink, such as a TPIU.

A minimum of a single-entry Stimulus Port output buffer, that is shared by all the Stimulus port registers, must be provided. The size of the output buffer is IMPLEMENTATION DEFINED. When the Stimulus Port output buffer is full, a write to a Stimulus Port is ignored, and an overflow packet is generated.

A Stimulus Port read indicates the output buffer status. The output buffer status reads return “full” when ITMENA or the Stimulus Port’s enable bit is clear (the port is disabled). ITMENA is cleared by a power-on reset.

Note

To ensure system correctness, a software polling scheme can use exclusive accesses to manage Stimulus Port writes with respect to the Stimulus Port output buffer status. Software must test the status by reading from the Stimulus Port that it intends to write.

All ITM registers can be read by unprivileged (User) and privileged code at all times. Privileged write accesses are ignored unless ITMENA is set. Unprivileged write accesses to the Trace Control and Trace Privilege registers are always ignored. Unprivileged write accesses to the Stimulus Port and Trace Enable registers are allowed or ignored according to the setting in the Trace Privilege Register. Trace Enable registers are byte-wise enabled for user access, according to the Trace Privilege Register setting.

Timestamp support

Timestamps provide information on the timing of event generation with respect to their visibility at a trace output port. The timestamp counter size, clock frequency, and whether the timestamp reference clock is synchronous or asynchronous with respect to the core clock are IMPLEMENTATION DEFINED. Timestamp packets are generated within the following constraints:

- Timestamp packets support a maximum count field of 28 bits. The transmitted packet should compress leading zeroes and transmit the minimum size packet required to support the timestamp value.
- Timestamp packet generation is enabled by the TSENA bit in the Trace Control Register (see *Trace Control Register – ITM_TCR (0xE0000E80)* on page C1-31). Synchronous versus Asynchronous operation is determined by the SWOENA control bit.
- Timestamp operation is defined as differential. A timestamp packet is generated when an event is posted to its associated FIFO and the timestamp counter is non-zero.
- The timestamp counter is cleared when a timestamp packet is generated.
- Synchronous configuration is selected by setting the TSENA bit and clearing the SWOENA bit in the Trace Control Register (see *Trace Control Register – ITM_TCR (0xE0000E80)* on page C1-31).
- Asynchronous configuration is selected by setting the TSENA and SWOENA bits in the Trace Control register (see *Trace Control Register – ITM_TCR (0xE0000E80)* on page C1-31).

  The relationship between the timestamp reference clock and an asynchronous TPIU port (see *Trace Port Interface Unit (TPIU)* on page C1-57) is fixed by the effective lineout clock rate of the asynchronous port, the rate dependent on the output encoding scheme (NRZ or Manchester).
- When asynchronous configuration is selected, the timestamp counter is held in reset (cleared) while the asynchronous interface is idle. This means that a timestamp is not generated on the first packet after idle on an asynchronous interface.
- When synchronous configuration is selected, it is IMPLEMENTATION DEFINED if timestamps are disabled in Debug state.

  Note

  ARM recommends that the generation of timestamps is disabled in Debug state.
- If the timestamp counter overflows, an overflow packet is generated and a timestamp packet is transmitted at the earliest opportunity (can be a value other than zero if delayed due to higher priority trace packets).

A timestamp packet can be generated and appended to a single event packet, or a stream of back-to-back packets where multiple events generate a packet stream with no idle time. Timestamp status information is merged with the timestamp packets to indicate if the timestamp packet transfer is delayed by the FIFO, or if there is a delay in the associated event packet transfer to the output FIFO. The timestamp count continues until it can be sampled and delivered in a packet to the FIFO.

Timestamp packets are combined with synchronization packets (see below) plus the ITM and Data Watchpoint and Trace (DWT, see *Data Watchpoint and Trace (DWT) support* on page C1-33) event packets according to the packet protocol prioritization rules described in *Multiple Source Arbitration* on page AppxE-7.

Synchronization support

Synchronization packets are independent of timestamp packets. They are used to recover bit to byte alignment information. The packets are required on synchronous TPIU ports that are dedicated to an ARMv7-M core or in complex systems where multiple trace streams are formatted into a single output. See *CoreSight Architecture Specification* for more details on synchronization and trace formatting. When enabled, synchronization packets are emitted on a regular basis and can be used as a system heartbeat. Synchronization packets should be disabled on asynchronous TPIU ports.

Synchronization packets are enabled by the SYNCENA bit in the Trace Control Register (see *Trace Control Register – ITM_TCR (0xE0000E80)* on page C1-31). The frequency of generation (the TPIU output heartbeat) is controlled by the SYNCTAP bitfield in the DWT Control Register (see *Control Register (DWT_CTRL)* on page C1-48).

C1.7.2 Register support for the ITM

Table C1-13 lists the ITM registers and their address offset. The base address for the ITM is `0xE0000000`.

Table C1-13 ITM registers

Address offset	Type (Read/Write)	Name	Notes
`0xE0000000`	R/W	ITM_STIM0	Stimulus Port #0
…	…	…	... word aligned locations between offsets 0x000 and 0x3FC (ports #0 to #255)
`0xE00003FC`	R/W	ITM_STIM255	Stimulus Port #255
`0xE0000E00`	R/W	ITM_TER0	Trace Enable ports #0 to #31 (Instrumentation)
…	…	…	
`0xE0000E1C`	R/W	ITM_TER7	Trace Enable ports #224 to #255
`0xE0000E40`	R/W	ITM_TPR	Trace Privilege (Instrumentation)

Table C1-13 ITM registers (continued)

Address offset	Type (Read/Write)	Name	Notes
0xE0000E80	R/W	ITM_TCR	Trace Control (Instrumentation)
0xE0000FB4[a]	RO		See Appendix B *ARMv7-M infrastructure IDs*, Table B-2 on page AppxB-3
0xE0000FD0			ID space: See Appendix B *ARMv7-M infrastructure IDs* for more information.
…			
0xE0000FFC			

[a] CoreSight compliance for the lock status register is required.

Stimulus Portx Register - ITM_STIM[255:0] (0xE0000000 to 0xE00003FC)

These registers provide the interface for generating instrumentation messages.

Table C1-14 Stimulus Port Register: STIMx

Bits[a]	R/W	Name	Function
[31:0]	W	STIMULUS	Data write to the Stimulus Port FIFO for forwarding as a software event packet. The write is ignored if the Stimulus Port is disabled by the Trace Enable Register.
[31:1]	R		RAZ
[0]	R	FIFOREADY	1: Stimulus Port FIFO can accept at least one word 0: Stimulus Port FIFO full. RAZ when the Stimulus Port is disabled by the Trace Enable Register.

[a] Bits read as UNKNOWN and write accesses are ignored when DEMCR.TRCENA == 0.

Trace Enable Register - ITM_TER[7:0] (0xE0000E00 to 0xE0000E1C)

This register provides individual mask bits for the stimulus registers.

Table C1-15 Transfer Enable Register: TER

Bits	R/W	Name	Function
[31:0]	R/W	STIMENA	Stimulus Port #N is enabled when bit STIMENA[N] is set. Unused bits are RAZ/WI. The register is cleared on Power-up reset.

User writes are ignored if the corresponding PRIVMASK bit is set.

Trace Privilege Register - ITM_TPR (0xE0000E40)

This register allows an operating system to control which stimulus ports can be accessed by User code.

Table C1-16 Trace Privilege Register: TPR

Bits	R/W	Name	Function
[31:0]	R/W	PRIVMASK	Bit mask to enable unprivileged (User) access to ITM stimulus ports. Bit [N] of PRIVMASK controls stimulus ports 8N to 8N+7. 0: User access allowed to stimulus ports 1: Privileged access only to stimulus ports Unused bits are RAZ/WI. The register is cleared on Power-up reset.

Trace Control Register – ITM_TCR (0xE0000E80)

Trace Control Register is used to configure and control transfers through the ITM interface.

Table C1-17 Trace Control Register: TCR

Bits	R/W	Name	Function
[31:24]	-		Reserved
[23]	RO	BUSY	Set when ITM events present and being drained
[22:16]	R/W	TraceBusID	Optional identifier for multi-source trace stream formatting. If multi-source trace is in use, this field must be written with a non-zero value. See *CoreSight Architecture Specification* for more details.
[15:10]			Reserved
[9:8]	R/W	TSPrescale	Timestamp prescaler, used with the trace packet reference clock. The reference clock source is selected by SWOENA. 00: no prescaling 01: divide by 4 10: divide by 16 11: divide by 64 These bits are cleared on Power-up reset.
[7:5]	-		Reserved

Table C1-17 Trace Control Register: TCR (continued)

Bits	R/W	Name	Function
[4]	R/W	SWOENA	Enables asynchronous-specific usage model for timestamps (when TSENA ==1) 0: mode disabled. Timestamp counter uses system clock from the core and counts continuously. 1: Timestamp counter uses lineout (data related) clock from TPIU interface. The timestamp counter is held in reset while the output line is idle.
[3]	R/W	TXENA[a]	Enable hardware event packet emission to the TPIU from the DWT. This bit is cleared on Power-up reset.
[2]	R/W	SYNCENA	Enable synchronization packet transmission for a synchronous TPIU. Note: DWT_CTRL.SYNCTAP must be configured for the correct synchronization speed. This bit is cleared on Power-up reset.
[1]	R/W	TSENA	Enable differential timestamps. Timestamp behavior is qualified by SWOENA. This bit is cleared on Power-up reset.
[0]	R/W	ITMENA	Enable ITM. This is the master enable and must be set to allow writes to all ITM registers, including the control register. This bit is cleared on Power-up reset.

[a] This bit was formerly known as DWTENA. The name was changed to better describe the limited scope of the signal.

The CoreSight compliant TraceBusID field is required in systems supporting multiple trace streams, for example an ARMv7-M core with ETM. This is the ID used to identify the ITM/DWT trace stream. The ID to identify the ETM trace stream is programmed in the ETM traceBusID register (see *Embedded Trace macrocell Architecture Specification*). To avoid trace stream corruption, the ITM must be disabled (ITMENA clear) and the BUSY bit polled until it is clear (reads zero) before the TraceBusID is modified.

C1.8 Data Watchpoint and Trace (DWT) support

The Data Watchpoint and Trace (DWT) component can provide the following features:

- PC sampling, two forms are supported:
  - PC sample trace output as a result of a cycle count event or DWT function match
  - external PC sampling using a PC sample register.
- comparators to support:
  - watchpoints – enters Debug state or takes a DebugMonitor exception
  - data tracing
  - signalling for use with an external resource, for example an ETM
  - cycle count matching.
- exception trace support
- profiling counter support.

See *Debug support in ARMv7-M* on page C1-10 for information on how to determine the level of DWT support in a given device.

C1.8.1 Theory of operation

Apart from exception tracing and an external agent PC sampling feature (DWT_PCSR, see *Program counter sampling support* on page C1-36), DWT functionality is counter or comparator based. Supported features can be determined from the debug ROM table, DEMCR.TRCENA master enable bit and feature availability bits in the DWT_CTRL register, see *Debug support in ARMv7-M* on page C1-10.

Exception tracing and counter control is provided by the DWT Control Register (see *Control Register (DWT_CTRL)* on page C1-48).

Watchpoint and data trace support use a set of compare, mask and function registers (see *Comparator Register (DWT_COMPx)* on page C1-53, *Mask Register (DWT_MASKx)* on page C1-53, and *Function Register (DWT_FUNCTIONx)* on page C1-54 for details).

DWT generated events result in one of four actions:

- generation of a Hardware Source packet. Packets are generated and combined with other event, control and timestamp packets according to the packet protocol described in Appendix E *Debug ITM and DWT packet protocol*.

  Note

  ~~Packet protocol support requires cycle count support (DWT_CTRL.TRCCNT == ’0’ && DWT_CRTL.CYCCNT == ’0’)~~

- a core halt – entry to Debug state
- a DebugMonitor exception

- generation of a CMPMATCH[N] signal as a control input to an external debug resource.

Note

DWT hardware event packet transmission is enabled in the ITM block (Trace Control Register), which also controls the timestamp support. For timestamp provision, the ITM stimulus ports can be disabled. However, the ITMENA and TSENA in the ITM Trace Control Register must be set to provide a timestamp capability.

Exception trace support

Exception tracing is enabled using the EXCTRCENA bit in the DWT_CTRL register. When the bit is set, the DWT emits an exception trace packet under the following conditions:

- exception entry (from Thread mode or pre-emption of thread or handler).
- exception exit when exiting a handler with an EXC_RETURN vector. See *Exception return behavior* on page B1-25.
- exception return when re-entering a pre-empted thread or handler code sequence.

Cycle counter and PC sampling support

The cycle count register (DWT_CYCCNT (or CYCCNT), see *Cycle Count Register (DWT_CYCCNT)* on page C1-49) is a 32-bit register. The register has five architected tap points:

- bit [6] or bit [10], as selected by the CYCTAP bit in the DWT_CTRL register, is used to clock a 4-bit post-scalar counter (POSTCNT).
- bit [24], bit [26] or bit [28], as selected by the SYNCTAP bits in the DWT_CTRL register, is used as an input to the ITM block to generate synchronization packets (see *Trace Control Register – ITM_TCR (0xE0000E80)* on page C1-31).

The interface to POSTCNT, along with control bits for CYCCNT, are in the DWT_CTRL register. The POSTPRESET field provides a reload value for POSTCNT.

CYCCNT is enabled and POSTCNT is preloaded from POSTPRESET when CYCCNTENA is set.

CYCCNT should be cleared before it is enabled.

When POSTCNT is non-zero, transitions (0 → 1 or 1 → 0) on the tap port selected by CYCTAP cause POSTCNT to decrement. When POSTCNT is zero and the selected tap port transitions, the action which occurs is as described in Table C1-18.

Table C1-18 Cycle count event generation

DWT_CTRL control bit			
CYCCNTENA	**PCSAMPLENA**	**CYCEVTENA**	**Action[a]**
0	x	x	CYCCNT (and POSTCNT) disabled
1	0	0	Events disabled, POSTPRESET => POSTCNT
1	0	1	An event packet with the Cyc bit set is emitted, see *Event Packets – Discriminator ID0* on page AppxE-8 POSTPRESET => POSTCNT
1	1	~~x~~	PC sample packet emitted, see *PC Sample Packets – Discriminator ID2* on page AppxE-9 POSTPRESET => POSTCNT

[a] POSTCNT == 0 and a count event occurs (the selected tap port of CYCCNT transitions 0 → 1 or 1 → 0)

Profiling counter support

In addition to CYCCNT and POSTCNT, the following 8-bit event counters are defined:

- DWT_CPICNT: the general counter for instruction cycle count estimation. The counter increments on any additional cycles (the first cycle is not counted) required to execute multi-cycle instructions except those recorded by DWT_LSUCNT. The counter also increments on any instruction fetch stalls.
- DWT_LSUCNT: the load-store count. The counter increments on any additional cycles (the first cycle is not counted) required to execute multi-cycle load-store instructions.
- DWT_FOLDCNT: the folded instruction count. The counter increments on any instruction that executes in zero cycles.
- DWT_EXCCNT: the exception overhead counter. The counter increments on any cycles associated with exception entry and return.
- DWT_SLEEPCNT: the sleep overhead counter. The counter increments on any cycles associated with power saving, initiated by the `WFI` or `WFE` instructions, or by use of the ~~SLEEPONEXIT control~~ feature. See *Power management* on page B1-48 for more details.

All 8-bit counters have separate enable bits in the DWT_CTRL register, and each counter is initialized to zero when it is enabled. The counters wrap when they overflow (every 256th cycle counted), and have counting suppressed when they are disabled. Counter overflow causes an event packet (see *Event Packets – Discriminator ID0* on page AppxE-8) to be emitted by the DWT, with the appropriate counter flag(s) set.

Profiling counter accuracy

The counters are designed to provide approximately accurate performance count information, but in order to keep the implementation and validation cost low, a reasonable degree of inaccuracy in the counts is acceptable. There is no exact definition of “reasonable degree of inaccuracy”, but the following guidelines should be followed.

Under normal operating conditions, the counters should present an accurate value of the overall system count:

- While the counters can be used with halting debug, they are primarily intended for non-intrusive operation. Entry or exit from Debug state can be a source of inaccuracy. Counters will not increment when halted. The overhead associated with STEP and RUN commands from and to the halt condition in Debug state is IMPLEMENTATION DEFINED.
- The intention is that where an instruction is used to enter or exit exceptions and sleep state (for example `SVC` and `WFI`), the cycle count associated with the instruction is minimal, with the balance of cycles associated with the exception overhead. The exact division is IMPLEMENTATION DEFINED.
- In superscalar implementations FOLD counts can be very high, affecting profiling statistics. Profile data validity will generally improve when aggregated over longer timeframes with large data and instruction working sets.

The permissibility of inaccuracy will limit the possible uses of the performance counters. In particular, the point in a pipeline where the performance counter is incremented is not defined relative to the point at which a read of the performance counters is made, so allowing for some imprecision due to pipelining effects. Counter size and the event generation model are designed primarily for non-intrusive operation, where the information is traced, processed and analysed remotely, and not subject to the system overhead of software reads and processing on the core itself.

Implementations should document any particular scenarios where significant inaccuracies are expected.

Program counter sampling support

The DWT Program Counter Sampling Register (DWT_PCSR) is an IMPLEMENTATION DEFINED option in ARMv7-M. The register is defined such that it can be accessed by a debugger without changing the behavior of any code currently executing on the device. This provides a mechanism for coarse-grained non-intrusive profiling of code executing on the core. The DWT_PCSR is a word-accessible read-only register, writes to the register are ignored. Byte or halfword reads are UNPREDICTABLE. When the register is read it returns one of the following:

- the address of an instruction *recently executed* by the core

- 0xFFFFFFFF if implemented and the processor is in Debug state, or in a state and mode where non-invasive debug is not permitted
- RAZ/WI if not implemented.

Note

There is no architectural definition of *recently executed*. The delay between an instruction being executed by the core and its address appearing in the DWT_PCSR is not defined. For example, if a piece of code reads the DWT_PCSR of the processor it is running on, there is no guaranteed relationship between the program counter for that piece of code and the value read. The DWT_PCSR is intended for use only by an external agent to provide statistical information for code profiling. Read accesses made to the DWT_PCSR directly by the ARM core can return an UNKNOWN value.

A debug agent should not rely on a return value of 0xFFFFFFFF to indicate that the core is halted. The S_HALT bit in the Debug Halting Control and Status Register should be used for this purpose.

The value read always references a *committed instruction*, where a committed instruction is defined as an instruction which is both fetched and committed for execution. It is IMPLEMENTATION DEFINED whether instructions that do not pass their condition codes are considered as committed instructions. ARM recommends that these instructions are treated as committed instructions.

If DWT_PCSR is implemented, it must be possible to sample references to branch targets. It is IMPLEMENTATION DEFINED whether references to other instructions can be sampled. ARM recommends that a reference to any instruction can be sampled.

The branch target for a conditional branch that fails its condition code is the instruction that follows the conditional branch instruction.

An implementation must not sample values that reference instructions that are fetched but not committed for execution. A read access from the DWT_PCSR returns an UNKNOWN value when the TRCENA bit in the Debug Exception and Monitor Control Register is clear.

The DWT_PCSR is not affected by the PCSAMPLENA bit in the DWT Control Register.

Comparator support

The DWT_COMPx, DWT_MASKx, and DWT_FUNCTIONx register sets provide the programming interface for the type of match to perform, and the action to take on a match. The number of register sets supported is IMPLEMENTATION DEFINED and can be determined by reading the NUMCOMP field in the DWT_CTRL register.

The types of match that can be performed are:

- watchpoint data address matching
- watchpoint data value matching (option within DWT)
- instruction address (PC) matching
- cycle count matching.

DWT_COMPx contains the reference value (COMP) for the comparator. This value is compared against an input value to determine a match.

For address matching (watchpoint data or instruction address) the input value is masked. The number of input value bits masked (MASK) is defined in DWT MASKx[4:0]. The maximum address mask range supported (up to 2GB) is IMPLEMENTATION DEFINED.

DWT_FUNCTIONx defines the matching conditions and control information for the comparator inputs and what information/action is returned on a match. One of three types of action can be taken on successful matches:

- Return sampled information in a hardware source packet through the ITM - see *Hardware Source Packet* on page AppxE-6 and *DWT packet formats* on page AppxE-8. Packets will be generated and transmitted where transmit FIFO space exists, otherwise an overflow packet (see *Overflow Packet* on page AppxE-3) will be generated where appropriate for each source to indicate packet loss.
- Halt and enter Debug state. The address of the next instruction to execute on a halt is IMPLEMENTATION DEFINED. A halt is asynchronous with respect to the instruction which caused it.
- Generate a CMPMATCH[N] event. CMPMATCH[N] events are for use external to the DWT block

Where a multiple match occurs, a HALT or CMPMATCH[N] match action will always be generated. Event packet generation on a multiple match is UNPREDICTABLE. An overflow packet is generated when a DWT event cannot create a DWT packet, because the DWT output buffer is full.

Table C1-19 summarizes the control bit settings in DWT_FUNCTIONx and how they relate to the other DWT registers.

Table C1-19 DWT register set feature summary

DWT_FUNCTIONx[bits][a]							DWT_MASKx	DWT_COMPx
[19:12]	[11:10]	[9]	[8]	[7]	[5]	[3:0]/Action		reference input
SBZ	SBZ	SBZ	0	0	x	See Table C1-20 on page C1-39	Mask_value	Address[b]
SBZ	SBZ	SBZ	0	1[c]	SBZ	See Table C1-21 on page C1-41	SBZ	Cycle count
LinkAddr()[d]	Dsize	x[e]	1[f]	0	SBZ	See Table C1-22 on page C1-42	SBZ	Data value[g]
x	x	x	1	1	x	UNPREDICTABLE	x	x

a. DWT_FUNCTION bit field names (see *Function Register (DWT_FUNCTIONx)* on page C1-54 for more information):

```
LNKADDR1:   bits [19:16]   LNKADDR0: bits [15:12]   LNK1ENA: bit [9]
DATAVMATCH: bit [8]        CYCMATCH: bit [7]        EMITRANGE: bit [5]
FUNCTION:   bits [3:0]
```

b. The address of the executing instruction or data access.

c. Supported on DWT_FUNCTION0 only. For other comparators bit [7] is RAZ/WI.

d. Two 4-bit fields providing pointers to 1 or 2 linked address comparators. If LNK1ENA == 0, only LNKADDR0 support is implemented. See *LinkAddr() support* on page C1-43 for more information.

e. See *DWT_FUNCTIONx* on page C1-54.

f. Optional support, otherwise RAZ/WI.

g. Word and halfword values are in little endian data format. A halfword value must be replicated in bits [15:0] and bits [31:16]. A byte value must be replicated in all (four) byte lanes.

Table C1-20 General DWT function support

DWT_FUNCTIONx			Comparator		Function Description/Action
(DATAV:CYC)MATCH[a] Bits [8:7]	EMITRANGE Bit [5]	FUNCTION Bits [3:0]	Input[b]	Access	Match(Input, COMP) == TRUE
00	x	0000			disabled or LinkAddr()[c] support
00	0	0001	Daddr	R/W	Sample[d][e] PC
00	1	0001	Daddr	R/W	Sample Daddr[15:0]
00	0	0010	Daddr	R/W	Sample data
00	1	0010	Daddr	R/W	Sample Daddr[15:0] + data
00	0	0011	Daddr	R/W	Sample PC + data
00	1	0011	Daddr	R/W	Sample Daddr[15:0] + data
00	x	0100	Iaddr	-	PC watchpoint event[f][g]
00	x	0101	Daddr	RO[h]	Watchpoint event (optional)
00	x	0110	Daddr	WO	Watchpoint event (optional)
00	x	0111	Daddr	R/W	Watchpoint event[g]
00	x	1000	Iaddr	-	Generate CMPMATCH[N][i] event
00	x	1001	Daddr	RO	Generate CMPMATCH[N] event (optional)
00	x	1010	Daddr	WO	Generate CMPMATCH[N] event (optional)
00	x	1011	Daddr	R/W	Generate CMPMATCH[N] event

Table C1-20 General DWT function support (continued)

DWT_FUNCTIONx			Comparator		Function Description/Action
(DATAV:CYC)MATCH[a] Bits [8:7]	EMITRANGE Bit [5]	FUNCTION Bits [3:0]	Input[b]	Access	Match(Input, COMP) == TRUE
00	0	1100	Daddr	RO	Sample data (optional)
00	1	1100	Daddr	RO	Sample Daddr[15:0] (optional)
00	0	1101	Daddr	WO	Sample data (optional)
00	1	1101	Daddr	WO	Sample Daddr[15:0] (optional)
00	0	1110	Daddr	RO	Sample PC + data (optional)
00	1	1110	Daddr	RO	Sample Daddr[15:0] + data (optional)
00	0	1111	Daddr	WO	Sample PC + data (optional)
00	1	1111	Daddr	WO	Sample Daddr[15:0] + data (optional)
01	x	xxxx			See Table C1-21 on page C1-41
1x	x	xxxx			See Table C1-22 on page C1-42

a. Shortform of DATAVMATCH:CYCMATCH.

b. Daddr: data access address match. Iaddr: instruction address match.

c. See *LinkAddr() support* on page C1-43 for more details. In the LinkAddr() case, the input is Daddr, the data access address.

d. This is the matched address of the executing instruction.

e. Sampled information is emitted as hardware event packets. See *DWT packet formats* on page AppxE-8 for packet format information. The behavior of this feature is UNPREDICTABLE if DWT_CTRL.NOTRCPKT == 1.

f. PC watchpoints can match on a range of addresses. Breakpoints only match on a specific instruction address.

g. DebugMonitor exception or Halt. The address of the next instruction to execute on a halt (DebugReturnAddress~~(~~) is IMPLEMENTATION DEFINED, see Table C1-10 on page C1-22~~)~~.

h. Support of explicit watchpoint read and write functionality is IMPLEMENTATION DEFINED.

i. CMPMATCH[N] events are for use external to the DWT block. The behavior of this feature is UNPREDICTABLE if DWT_CTRL.NOEXTTRIG == 1.

A watchpoint event is an asynchronous event with respect to the instruction which caused it. Watchpoints are treated as interrupts in the exception model.

Support of explicit watchpoint read and watchpoint write functionality can be determined by writing the appropriate value to DWT_FUNCTIONx[FUNCTION] and reading back the result. Read-only or write-only watchpoint requests convert to read/write watchpoints where the requested feature is not supported.

Table C1-21 DWT comparator support for CYCCNT

DWT_FUNCTION0[a]		Function Description/Action
(DATAV:CYC)MATCH[b] Bits [8:7]	FUNCTION Bits [3:0]	Match(CYCCNT, COMP) == TRUE
00	xxxx	See Table C1-20 on page C1-39
1x	xxxx	See Table C1-22 on page C1-42
01	0000	Disabled
01	0001	Sample[c] PC
01	001x	UNPREDICTABLE
01	0100	Watchpoint event
01	0101	UNPREDICTABLE
01	011x	UNPREDICTABLE
01	1000	CMPMATCH0[d] event generated
01	1001	UNPREDICTABLE
01	101x	UNPREDICTABLE
01	11xx	Reserved

a. DWT_FUNCTION0 only. For x > 0, DWT_FUNCTIONx[7] is RAZ/WI.

b. Shortform of DATAVMATCH:CYCMATCH.

c. This is the matched address of the executing instruction. Sampled information is emitted as hardware event packets. See *DWT packet formats* on page AppxE-8 for packet format information. The behavior of this feature is UNPREDICTABLE if DWT_CTRL.NOTRCPKT == 1.

d. CMPMATCH events are for use external to the DWT block. The behavior of this feature is UNPREDICTABLe if DWT_CTRL.NOEXTTRIG == 1

Table C1-22 DWT comparator support for data matching

DWT_FUNCTIONx		Comparator		Function Description/Action
(DATAV:CYC)MATCH Bits[8:7]	FUNCTION Bits [3:0]	Input	Access	Match(Input, COMP) == TRUE
00	xxxx			See Table C1-20 on page C1-39
01	xxxx			See Table C1-21 on page C1-41
11	xxxx			UNPREDICTABLE
10	0000			Disabled
10	00x1			UNPREDICTABLE
10	001x			UNPREDICTABLE
10	0100			UNPREDICTABLE
10	0101	Data[a]	RO[b]	Watchpoint event[c] (optional)
10	0110	Data	WO	Watchpoint event (optional)
10	0111	Data	R/W	Watchpoint event
10	1000			UNPREDICTABLE
10	1001	Data	RO	CMPMATCH [N][d] event generated (optional)
10	1010	Data	WO	CMPMATCH [N] event generated (optional)
10	1011	Data	R/W	CMPMATCH [N] event generated
10	11xx			UNPREDICTABLE

[a] For details on data matching see *Comparator support - watchpoint data value matching* on page C1-45.

[b] Support of explicit watchpoint read and write functionality is IMPLEMENTATION DEFINED.

[c] The address of the next instruction to execute on a halt (DebugReturnAddress~~()~~, see Table C1-10 on page C1-22 on page page C1-22) is IMPLEMENTATION DEFINED. A halt is imprecise with respect to the instruction which caused it.

[d] CMPMATCH events are for use external to the DWT block. The behavior of this feature is UNPREDICTABLE if DWT_CTRL.NOEXTTRIG == 1.

Support of explicit watchpoint read and watchpoint write functionality can be determined by writing the appropriate value to DWT_FUNCTIONx[FUNCTION] and reading back the result. Read-only or write-only watchpoint requests convert to read/write watchpoints where the requested feature is not supported.

LinkAddr() support

Data value matching can be gated by a configurable data access address match using a comparator associated with another register set.

DWT_FUNCTION[N] is defined as the set of registers DWT_FUNCTIONx, DWT_COMPx and DWT_MASKx where x = N. Up to two comparators associated with other register sets (x != N) can be configured using the LNK1ENA, DATAVADDR0 and DATAVADDR1 fields in DWT_FUNCTION[N]. See *Function Register (DWT_FUNCTIONx)* on page C1-54 for more details.

See *Comparator support - data address matching* on page C1-44 for details on link address matching.

Where LNK1ENA == 1, a single link address can be configured by configuring both to the same link address comparator or disabling one of them by configuring the feature as not used.

Comparator support - cycle count matching

Cycle count matching is only supported on DWT_COMP0. For all other register sets, the CYCMATCH field is RAZ/WI. Non-zero values of DWT_MASK0 are UNPREDICTABLE.

Note

The DWT_CRTRL.NOCYCCNT bit field is set when the cycle counter is not supported. In this case validCYCMATCH is always false.

The comparator behavior is defined as follows:

```
boolean validCYCMATCH; // conditions for selecting CYCCNT from
                       // ... configuration of the DWT_FUNCTION0 register
if validCYCMATCH then
    match = (CYCCNT<31:0> == DWT_COMP0<31:0>);
    return match;
```

Comparator support - instruction address matching

Instruction address matching is supported on all comparators. DWT_COMPx must be halfword aligned, and the mask register must be applied by software to the comparator reference value before it is written to DWT_COMPx, otherwise the comparator operation is UNPREDICTABLE.

Instruction address matches on `NOP` or `IT` instructions are UNPREDICTABLE with respect to whether the event occurs or not.

The comparator behavior is defined as follows:

```
// InstructionAddressMatch()
// =========================

boolean InstructionAddressMatch(integer N, bits(32) Iaddr)

    if DWT_FUNCTION[N]<3:0> == ’0100’ then                // condition for selecting Iaddr
        // UNPREDICTABLE if COMP does not meet alignment and masking conditions
        mask = ZeroExtend(Ones(UInt(DWT_MASK[N]<4:0>)), 32);
        if !IsZero(DWT_COMP[N] & mask) then UNPREDICTABLE;
        match = (Iaddr & NOT(mask) == DWT_COMP[N]);
    else
        match = FALSE;
    return match;
```

Comparator support - data address matching

Where a data value comparator is linked using the LinkAddr() support, an address match occurs if either the first or second linked data address comparator matches. If LinkAddr() is configured as not used, a data value match can occur on any data access.

For data address compares, the implementation must address match all memory accesses, where the range of watched addresses lie between the start address of the transaction and the next word aligned address. It is IMPLEMENTATION DEFINED whether the comparison matches some or all unaligned memory accesses that access a watched location across a word boundary. Data address matching comparator behavior is defined as follows:

```
enumeration sizeofaccess (byte, halfword, word);
boolean validDaddr;       // conditions for selecting Daddr from
                          // ...configuraton of the relevant DWT_FUNCTIONx register
if validDaddr then
    match = DataAddressMatch(N, Daddr, Dsize);
    return match;



// DataAddressMatch()
// ==================

boolean DataAddressMatch(UInt N, bits(32) address, sizeofaccess size)

    // UNPREDICTABLE if the base compare address isn’t properly aligned
    mask = ZeroExtend(Ones(UInt(DWT_MASK[N]<4:0>)), 32);
    if !IsZero(comp_start & mask) then UNPREDICTABLE;

    // compute start and end addresses of compared region
    comp_start = DWT_COMP[N]<31:0>;
    comp_end = comp_start + mask;


    // compute start and end addresses of access
    access_start = address;
    case size of
        when word
            access_end = access_start + 3;
        when halfword
            access_end = access_start + 1;
        when byte
            access_end = access_start;

    // Implementations can terminate matching on a word aligned address boundary > access_start
    if (IMPLEMENTATION DEFINED condition) then
        temp = ((access_end + 1) & ~3) - 1;
       if (temp > access_start) then
           access_end = temp;

    match = (access_start >= comp_start && access_start <= comp_end) ||
            (access_end   >= comp_start && access_end   <= comp_end) ||
            (access_start <= comp_start && access_end   >= comp_end);
    return match;
```

Comparator support - watchpoint data value matching

Support of data value matching is an architecture option. It is IMPLEMENTATION DEFINED how many of the available comparators support it. Non-zero values of DWT_MASK with data value matching are UNPREDICTABLE.

For halfword and byte data value matches, software must replicate the data value in DWT_COMPx otherwise matching is UNPREDICTABLE. Values are stored in DWT_COMPx in little endian data format.

- For byte compares: DWT_COMPx[31:24], DWT_COMPx[23:16], DWT_COMPx[15:8], and DWT_COMPx[7:0] should be the same value
- For halfword compares: DWT_COMPx[31:16], and DWT_COMPx[15:0] should be the same value

Data value matching generates a match where the data value to be matched is the same as the data access value, or is a partial match where the size of the access is larger than that specified in the DATAVSIZE bit field.

Data value matching is gated by LinkAddr() - see *LinkAddr() support* on page C1-43. It is IMPLEMENTATION DEFINED whether the data value matching is exact on the data address. If matching is exact, a match is only generated if the data value in DWT_COMPx precisely matches the value in memory at the address specified in the linked address comparator DWT_COMPy. Inexact matches are permitted. The conditions which generate inexact matches are IMPLEMENTATION DEFINED.

For example, if DWT_FUNCTIONx.DATAVSIZE specifies a halfword, and linked address comparator DWT_COMPy specifies a halfword aligned address with DWT_MASKy specifying a 1-bit address mask, then an exact match would match only those accesses made where the halfword location at DWT_COMPy is accessed with the value DWT_COMPx. However, an inexact match can be generated for a word access which accesses either or both bytes of [DWT_COMPy…DWT_COMPy+1], and is such that bits [15:0], bits [23:8] or bits [31:16] of the access value match DWT_COMPx, even if this value is not the value in memory at DWT_COMPy.

Comparator behavior for inexact data matching is illustrated as follows:

```
boolean validVMATCH;    // conditions for selecting data value matching from
                        //  ... configuration of the relevant DWT_FUNCTIONx register

if validVMATCH then
    addrmatch1 = addrmatch2 = FALSE;

    doaddrmatch1 = (DWT_FUNCTION[N]<9> == '1' && UInt(DWT_FUNCTION[N]<19:16>) != N);
    addrmatch1 = doaddrmatch1 && DataAddressMatch(UInt(DWT_FUNCTION[N]<19:16>), Daddr, Dsize);
```

```
doaddrmatch2 = (UInt(DWT_FUNCTION[N]<15:12>) != N);
addrmatch2 = doaddrmatch2 && DataAddressMatch(UInt(DWT_FUNCTION[N]<15:12>), Daddr, Dsize);

bits(8*Dsize) data = MemU[Daddr, Dsize];
case DWT_FUNCTION[N]<11:10> of
    when '00'
        case Dsize of
            when 1
                data_match = (DWT_COMP[N]<7:0> == data<7:0>);
            when 2
                data_match = (DWT_COMP[N]<7:0> == data<7:0> ||
                              DWT_COMP[N]<7:0> == data<15:8>);
            when 4
                data_match = (DWT_COMP[N]<7:0> == data<7:0>   ||
                              DWT_COMP[N]<7:0> == data<15:8>  ||
                              DWT_COMP[N]<7:0> == data<23:16> ||
                              DWT_COMP[N]<7:0> == data<31:24>);
    when '01'
        case Dsize of
            when 1
                data_match = FALSE;
            when 2
                data_match = (DWT_COMP[N]<15:0> == data<15:0>);
            when 4
                data_match = (DWT_COMP[N]<15:0> == data<15:0>  ||
                              DWT_COMP[N]<15:0> == data<23:8>  ||
                              DWT_COMP[N]<15:0> == data<31:16>);
    when '10'
        case Dsize of
            when 1
                data_match = FALSE;
            when 2
                data_match = FALSE;
            when 4
                data_match = (DWT_COMP[N]<31:0> == data<31:0>);
    when '11'
        data_match = UNKNOWN;

if (doaddressmatch1 || doaddressmatch2) then
    match = (addrmatch1 || addrmatch2) && data_match;
else
    match = data_match;
return match;
```

Note

For the DWT register set feature summary, see Table C1-19 on page C1-38.

For address matching information, see *Comparator support - data address matching* on page C1-44.

C1.8.2 Register support for the DWT

The DWT is programmed using the registers described in Table C1-23.

Table C1-23 DWT register summary

Address	R/W	Name	Notes
`0xE0001000`	R/W	DWT_CTRL	Control register
`0xE0001004`	R/W	DWT_CYCCNT	Cycle count register
`0xE0001008`	R/W	DWT_CPICNT	Instruction cycle count event register
`0xE000100C`	R/W	DWT_EXCCNT	Exception overhead event register
`0xE0001010`	R/W	DWT_SLEEPCNT	Sleep overhead event register
`0xE0001014`	R/W	DWT_LSUCNT	Load-store overhead event register
`0xE0001018`	R/W	DWT_FOLDCNT	Folded instruction overhead event register
`0xE000101C`	RO	DWT_PCSR	PC sampling register
`0xE0001020`+X	R/W	DWT_COMP[N-1]	X is the value (N-1) << 4 where: 0 < N ≤ DWT_CTRL.NUMCOMP
`0xE0001024`+X	R/W	DWT_MASK[N-1]	X is the value (N-1) << 4 where: 0 < N ≤ DWT_CTRL.NUMCOMP
`0xE0001028`+X	R/W	DWT_FUNCTION[N-1]	X is the value (N-1) << 4 where: 0 < N ≤ DWT_CTRL.NUMCOMP
`0xE0001FB4`[a]	RO		See Appendix B *ARMv7-M infrastructure IDs*, Table B-2 on page AppxB-3
`0xE0001FD0`	RO		ID space: See Appendix B *ARMv7-M infrastructure IDs* for more information.
…			
`0xE0001FFC`			

[a] CoreSight compliance for the lock status register is required.

Control Register (DWT_CTRL)

Table C1-24 DWT_CTRL (0xE0001000)

Bits	R/W	Name	Reset Value	Function[a]
[31:28]	RO	NUMCOMP	IMP DEF	Number of comparators available. NUMCOMP == 0: no comparator support
[27]	RO	NOTRCPKT[b]	IMP DEF	When set, trace sampling and exception tracing are not supported
[26]	RO	NOEXTTRIG[c]	IMP DEF	When set, no **CMPMATCH[N]** support
[25]	RO	NOCYCCNT[d]	IMP DEF	When set, DWT_CYCCNT is not supported
[24]	RO	NOPRFCNT[e]	IMP DEF	When set, DWT_FOLDCNT, DWT_LSUCNT, DWT_SLEEPCNT, DWT_EXCCNT, and DWT_CPICNT are not supported
[22]	R/W	CYCEVTENA	0b0	Used with PCSAMPLENA to control CYCCNT or PC sample event generation. See Table C1-18 on page C1-35 for details.
[21]	R/W	FOLDEVTENA	0b0	Enables Folded-instruction count event.
[20]	R/W	LSUEVTENA	0b0	Enables LSU count event.
[19]	R/W	SLEEPEVTENA	0b0	Enables Sleep count event.
[18]	R/W	EXCEVTENA	0b0	Enables Exception Overhead event.
[17]	R/W	CPIEVTENA	0b0	Enables CPI count event.
[16]	R/W	EXCTRCENA	0b0	Enables exception trace. This traces exception entry, exit and return (to a pre-empted handler or thread).
[12]	R/W	PCSAMPLENA	0b0	See CYCEVTENA.
[11:10]	R/W	SYNCTAP	UNKNOWN	Selects a synchronization packet rate. CYCCNTENA and ITM_TCR.SYNCENA must also be enabled for this feature. 00: Disabled. No synchronization packets. 01: Tap at CYCCNT bit 24 10: Tap at CYCCNT bit 26 11: Tap at CYCCNT bit 28 Synchronization packets (if enabled) generated on tap transitions (0 to1 or 1 to 0)

Table C1-24 DWT_CTRL (0xE0001000) (continued)

Bits	R/W	Name	Reset Value	Function[a]
[9]	R/W	CYCTAP	UNKNOWN	Selects a *tap* on the DWT_CYCCNT register. 0: Tap at CYCCNT bit 6 1: Tap at CYCCNT bit 10 Transitions generate a count event for POSTCNT.
[8:5]	R/W	POSTCNT	UNKNOWN	~~4-bit post-scalar counter used to extend the range of CYCCNT. Decrements when value > 1 on a count event (see CYCTAP)~~
[4:1]	R/W	POSTPRESET	UNKNOWN	Preset (reload) value for POSTCNT POSTPRESET => POSTCNT occurs on a count event when POSTCNT == 0.
[0]	R/W	CYCCNTENA	0b0	Enable CYCCNT, allowing it to increment and generate synchronization and count events. If DWT_CTRL.NOCYCCNT == 1, this bit is RAZ/WI.

a. For details of counter operation and event generation, see *Cycle counter and PC sampling support* on page C1-34

b. When the NOTRCPKT bit is set, DWT_FUNCTIONx[3:2] == '00', and DWT_FUNCTIONx[1:0] != '00', DWT comparator matching is UNPREDICTABLE. DWT_CRTRL bits [22,16,12] are reserved.
When the NOTRCPKT bit is clear, the CYCCNT feature is also required.

c. When the NOEXTTRIG bit is set, and DWT_FUNCTIONx[3:2] == '10', DWT comparator matching is UNPREDICTABLE.

d. If NOCYCCNT == 1, DWT_CTRL[22,12:0] are reserved and DWT_FUNCTION0[7] is RAZ/WI.

e. If NOPRFCNT == 1, DWT_CTRL[21:17] are reserved

Cycle Count Register (DWT_CYCCNT)

This counter is not supported (DWT_CYCCNT is RAZ/WI) if DWT_CTRL.NOCYCCNT == 1.

Table C1-25 DWT_CYCCNT (0xE0001004)

Bits	R/W	Name	Function
[31:0]	R/W	CYCCNT	32-bit, incrementing (up) cycle counter. When enabled, this counter counts the number of core cycles. Counting is suspended when the core is halted in Debug state. The counter is UNKNOWN on reset. CYCCNT wraps to 0 on overflow.

The counter can be used by applications and debuggers to measure elapsed execution time by taking the difference between sampled start and stop values within the 2^{32} clock tick dynamic range of the counter.

CPI Count Register (DWT_CPICNT)

This counter is not supported (DWT_CPICNT is RAZ/WI) if DWT_CTRL.NOPRFCNT == 1.

Table C1-26 DWT_CPICNT (0xE0001008)

Bits	R/W	Name	Function
[31:8]			Reserved
[7:0]	R/W	CPICNT	The base CPI counter. The counter increments on any additional cycles (the first cycle is not counted) required to execute multi-cycle instructions except those recorded by DWT_LSUCNT. The counter also increments on any instruction fetch stalls. An event is emitted when the counter overflows. This counter initializes to 0 when enabled (CPIEVTENA 0 => 1 transition).

Exception Overhead Count Register (DWT_EXCCNT)

This counter is not supported (DWT_EXCCNT is RAZ/WI) if DWT_CTRL.NOPRFCNT == 1.

Table C1-27 DWT_INTCNT (0xE000100C)

Bits	R/W	Name	Function
[31:8]			Reserved
[7:0]	R/W	EXCCNT	The exception overhead counter. This counter counts the total cycles spent in exception processing (entry stacking, return unstacking, pre-emption, etc). An event is emitted on counter overflow (every 256 cycles). This counter initializes to 0 when enabled (EXCEVTENA 0 => 1 transition).

Sleep Count Register (DWT_SLEEPCNT)

This counter is not supported (DWT_SLEEPCNT is RAZ/WI) if DWT_CTRL.NOPRFCNT == 1.

Table C1-28 DWT_SLEEPCNT (0xE0001010)

Bits	R/W	Name	Function
[31:8]			Reserved
[7:0]	R/W	SLEEPCNT	Sleep counter. This counts the total number of cycles that the processor is sleeping (initiated by `WFI`, `WFE`, Sleep on exit). An event is emitted on counter overflow (every 256 cycles). This counter initializes to 0 when enabled (SLEEPEVTENA 0 => 1 transition).

LSU Count Register (DWT_LSUCNT)

This counter is not supported (DWT_LSUCNT is RAZ/WI) if DWT_CTRL.NOPRFCNT == 1.

Table C1-29 DWT_LSUCNT (0xE0001014)

Bits	R/W	Name	Function
[31:8]			Reserved
[7:0]	R/W	LSUCNT	The load-store count. The counter increments on the additional cycles (the first cycle is not counted) required to execute all load-store instructions.An event is emitted on counter overflow (every 256 cycles). This counter initializes to 0 when enabled (LSUEVTENA 0 => 1 transition).

Folded-instruction Count Register (DWT_FOLDCNT)

This counter is not supported (DWT_FOLDCNT is RAZ/WI) if DWT_CTRL.NOPRFCNT == 1.

Table C1-30 DWT_FOLDCNT (0xE0001018)

Bits	R/W	Name	Function
[31:8]			Reserved
[7:0]	R/W	FOLDCNT	Folded-instruction counter. The counter increments on each instruction which takes 0 cycles. An event is emitted on counter overflow (every 256 cycles). This counter initializes to 0 when enabled (FOLDEVTENA 0 => 1 transition). If an implementation does not support instruction folding, this counter can be RAZ/WI.

Program Counter Sample Register (DWT_PCSR)

Table C1-31 DWT_PCSR (0xE000101C)

Bits	R/W	Name	Function
[31:0]	RO	EIASAMPLE	Executed Instruction Address sample value

The register value is UNKNOWN on reset. Bit [0] is RAZ and does not reflect instruction set state as is the case with similar functionality in other ARM architecture profiles.

Comparator Register (DWT_COMPx)

Table C1-32 DWT_COMPx

Bits	R/W	Name	Function
[31:0]	R/W	COMP	Reference value for comparison against CMPin. See *Comparator support* on page C1-37. The value is UNKNOWN on reset.

Mask Register (DWT_MASKx)

Table C1-33 DWT_MASKx

Bits	R/W	Name	Function
[31:5]	-		Reserved
[4:0]	R/W	MASK	The size of the ignore mask (0-31 bits) applied to address range matching. See *Comparator support* on page C1-37 for the usage model. The mask range is IMPLEMENTATION DEFINED. Writing all 1's to this field and reading it back can be used to determine the maximum mask size supported. The value is UNKNOWN on reset.

Function Register (DWT_FUNCTIONx)

This register controls the operation of the comparator DWT_COMPx.

Table C1-34 DWT_FUNCTIONx

Bits	R/W	Name	Function
[31:25]	-		Reserved
[24]	RO	MATCHED[a]	This bit is set when the associated comparator matches. It indicates that the operation defined by FUNCTION has occurred since the bit was last read. The bit is cleared on a read.
[23:20]	R/W		Reserved
[19:16]	R/W	DATAVADDR1[b]	When DATAVMATCH == ‘1’ and LNK1ENA == ‘1’:[c] value == x: DWT_COMPx used for data comparison value != x: Identity of a 2nd linked address comparator
[15:12]	R/W	DATAVADDR0[b]	When DATAVMATCH == ‘1’:[c] value == x: DWT_COMPx used for data comparison value != x: Identity of a linked address comparator
[11:10]	R/W	DATAVSIZE	Defines the size of the data in the associated COMP register for value matching: 00: byte 01: halfword 10: word 11: UNPREDICTABLE
[9]	RO	LNK1ENA	0: DATAVADDR1 not supported 1: DATAVADDR1 supported (enabled)
[8]	R/W	DATAVMATCH	Optional feature, otherwise RAZ/WI. 0: perform address comparison 1: perform data value compare See LNK1ENA, DATAVSIZE, LNKADDR0 and LNKADDR1 for related information.
[7]	R/W	CYCMATCH[d]	DWT_FUNCTION0 only, otherwise RAZ/WI. When set, DWT_COMP0 will compare against the cycle counter (DWT_CYCCNT).
[6]			Reserved

Table C1-34 DWT_FUNCTIONx (continued)

Bits	R/W	Name	Function
[5]	-	EMITRANGE[e]	When set, emit Daddr[15;0] in a hardware event packet according to Table C1-20 on page C1-39.
[4]	-		Reserved
[3:0]	R/W	FUNCTION	Select action on comparator match: `0000: disabled` or LinkAddr() - see *LinkAddr() support* on page C1-43. For non-zero values: If CYCMATCH clear, see Table C1-20 on page C1-39 If CYCMATCH set, see Table C1-21 on page C1-41 If DATAVMATCH set, see Table C1-22 on page C1-42 This field is reset to zero.

[a] See the footnotes in Table C1-24 on page C1-48 for details of comparator matching configurations affecting this bit that are UNPREDICTABLE.

[b] If DWT_FUNCx.DATAVMATCH == 1 and neither link address is configured all data transactions are tested for a data value match.

[c] otherwise the bit is ignored.

[d] Only applies if DWT_CTRL.NOCYCCNT == 0 to indicate cycle counting supported, otherwise the bit is RAZ/WI.

[e] Only applies if DWT_CTRL.NOTRCPKT == 0, otherwise the bit is RAZ/WI.

C1.9 Embedded Trace (ETM) support

ETM is an optional feature in ARMv7-M. Where it is supported, a Trace Port Interface Unit must be provided that is capable of formatting an output packet stream from the ETM and DWT/ITM packet sources.

An ETM implementation must be compliant with the ETM architecture v3.4 or a later version as defined in the *Embedded Trace Macrocell Architecture Specification*. The CMPMATCH[N] signals from the DWT block (see Table C1-20 on page C1-39 in *Comparator support* on page C1-37) are exported and available as control inputs to the ETM block. The associated TPIU implementation should be CoreSight compliant (see the *CoreSight Architecture Specification*) and align with the TPIU architecture for compatibility with ARM and other CoreSight compatible debug solutions.

TRCENA from the Debug Exception and Monitor Control Register (see *Debug Exception and Monitor Control Register (DEMCR)* on page C1-24) can be used as an enable signal for the ETM block.

Note

Enabling the ETM with TRCENA is IMPLEMENTATION DEFINED, and could be inappropriate where the ETM unit is a shared resource in a complex system.

C1.10 Trace Port Interface Unit (TPIU)

Hardware events from the DWT block and software events from the ITM block are multiplexed with timestamp information into a packet stream. Control and Configuration of the timestamp information and the packet stream is part of the DWT and ITM blocks. It is IMPLEMENTATION DEFINED whether the packets are made visible (requires pins or a trace buffer and access mechanism to be provided) or terminate within the core. Where a trace buffer is implemented, an Embedded Trace Buffer (ETB) solution should be CoreSight Architecture compliant as described in the *CoreSight Architecture Specification*.

External visibility requires an implementation to provide a Trace Port Interface Unit (TPIU). The ARMv7-M TPIU programmers' model includes support for an asynchronous Serial Wire Output (SWO) or a synchronous (single or multi-bit data path) trace port. The combination of the DWT/ITM packet stream and a SWO is known as a Serial Wire Viewer (SWV).

The minimum TPIU support for ARMv7-M provides an output path for a DWT/ITM generated packet stream of hardware and/or software generated event information. This is known as TPIU support for debug trace with the TPIU operating in pass-through mode.

For inclusion of an ETM, or other CoreSight compliant options, see *CoreSight Architecture Specification* and *Embedded Trace Macrocell Architecture Specification* for additional detail and options.

Synchronous trace ports can be supported in data path widths from 1 to 32 bits. Asynchronous serial ports can be supported in two options:

- A low speed asynchronous mode (NRZ encoding). This operates like a traditional UART.
- A medium-speed asynchronous mode (Manchester encoding).

An implementation can support both synchronous and asynchronous interfaces, the active interface selected by the Selected Pin Protocol Register as defined in Table C1-38 on page C1-59. It is recommended that both synchronous and asynchronous ports are provided for maximum flexibility with external capture devices. The reference clock for the synchronous port is generated internally, while the reference clock for the asynchronous port is generated from the effective lineout clock rate. A prescale counter is defined in the TPIU for the asynchronous port as part of the clock generation scheme for asynchronous operation.

C1.10.1 The TPIU Programmers' Model

This section defines the registers that must be present in an ARMv7-M TPIU for a minimum TPIU configuration. The register list is summarized in Table C1-35 on page C1-58.

TRCENA from the Debug Exception and Monitor Control register (see *Debug Exception and Monitor Control Register (DEMCR)* on page C1-24) can be used as an enable signal for the TPIU block.

Note

Enabling the TPIU with TRCENA is recommended in a minimal system but IMPLEMENTATION DEFINED. It could be inappropriate where the TPIU unit is a shared resource in a complex system.

Table C1-35 TPIU programmers' model overview

Address	Type	Reset Value	Name	Notes
`0xE0040000`	RO	IMP DEF	TPIU_SSPSR	Supported Synchronous Port Size
`0xE0040004`	R/W	IMP DEF	TPIU_CSPSR	Current Synchronous Port Size
`0xE0040010`	R/W	0x0	TPIU_ACPR	Asynchronous Clock Prescaler
`0xE00400F0`	R/W	IMP DEF	TPIU_SPPR	Selected Pin Protocol
`0xE0040FB4`[a]	RO	IMP DEF		See Appendix B *ARMv7-M infrastructure IDs*, Table B-2 on page AppxB-3
`0xE0040FC8`	RO	IMP DEF	TPIU_TYPE	
`0xE0040FD0`	RO			ID space: See Appendix B *ARMv7-M infrastructure IDs* for more information.
…				
`0xE0040FFC`				

[a] CoreSight compliance for the lock status register is required.

Supported Synchronous Port Sizes Register (TPIU_SSPSR, 0xE0040000)

Each bit location represents a supported port size. The bits set indicate the valid synchronous trace port sizes for the implementation.

Table C1-36 Supported Synchronous Port Sizes Register (0xE0040000)

Bits	R/W	Reset Value	Function
[31:0]	RO	IMP DEF	bit [N] == 0, trace port width of (N+1) not supported bit [N] == 1, trace port width of (N+1) supported

Current Synchronous Port Size Register (TPIU_CSPSR, 0xE0040004)

This has the same format as the Supported Port Sizes Register but only one bit is set (all others must be zero). Writing values with more than one bit set is UNPREDICTABLE. Writes to unsupported bits are UNPREDICTABLE.

On reset the register will default to the smallest supported port size.

Asynchronous Clock Prescaler Register (TPIU_ACPR, 0xE0040010)

This register is used to scale the baud rate of the Serial Wire Output. The asynchronous port reference clock is prescaled according to the value of bits [15:0]. When the register is written, the prescale counter is automatically updated (it is IMPLEMENTATION DEFINED whether it is preset and counts down or reset and counts up), affecting the baud-rate of the serial data output immediately.

If the register is updated while data is being transmitted, the affect on the output stream is UNPREDICTABLE and recovery is IMPLEMENTATION DEFINED.

Table C1-37 Asynchronous Clock Prescaler Register (0xE0040010)

Bits	R/W	Reset Value	Function
[31:16]	-		Reserved
[15:0]	R/W	UNKNOWN	Value used as a division ratio (baud rate prescaler). The available range is IMPLEMENTATION DEFINED with unused bits RAZ/WI. SWO output clock = Asynchronous_Reference_Clock/(value +1)

Selected Pin Protocol Register (TPIU_SPPR, 0xE00400F0)

This register selects which protocol to use for trace output. The supported values allowed within the register are determined by bits [11:9] of the TPIU Type Register (at address `0xE0040FC8`).

Table C1-38 Selected Pin Protocol Register (0xE00400F0)

Bits	R/W	Reset Value	Function
[31:2]	-		Reserved
[1:0]	R/W	IMP DEF	00: Synchronous Trace Port Mode 01: Asynchronous Serial Wire Output (Manchester) 10: Asynchronous Serial Wire Output (NRZ) 11: Reserved

If this register is changed whilst trace data is being output, the output behavior becomes UNPREDICTABLE and recovery is IMPLEMENTATION DEFINED.

TPIU Type Register (TPIU_TYPE, 0xE0040FC8)

This register defines the SWO supported options within the TPIU.

Table C1-39 TPIU Type Register (0xE0040FC8)

Bits	R/W	Function
[31:16]	RO	Reserved
[15:12]	RO	IMPLEMENTATION DEFINED
[11]	RO	Serial Wire Output (UART/NRZ) supported when the bit is set
[10]	RO	Serial Wire Output (Manchester encoding) supported when the bit is set
[9]	RO	RAZ, indicates that trace data and clock are supported
[8:6]	RO	Minimum output FIFO size for trace information (power of 2 bytes) For example, a value of 0b010 indicates at least a 4 byte output FIFO
[5:0]	RO	IMPLEMENTATION DEFINED

C1.11 Flash Patch and Breakpoint (FPB) support

The Flash Patch and Breakpoint (FPB) component can provide support for:

- remapping of specific literal locations from the Code region of system memory to an address in the SRAM region. See Table B3-1 on page B3-3 for information on address regions.
- remapping of specific instruction locations from the Code region of system memory to an address in the SRAM region. See Table B3-1 on page B3-3 for information on address regions.
- breakpoint functionality for instruction fetches.

The number of literal and instruction comparators are IMPLEMENTATION DEFINED and can be read from the FP_CTRL register (see *FlashPatch Control Register (FP_CTRL)* on page C1-64). The valid combinations of support are:

- no comparator support
- instruction comparator(s) with breakpoint support only
- instruction comparator(s) with breakpoint and remapping support
- a full feature set provided by instruction and literal comparator support.

Note

The FPB is not restricted to debug use only. The FPB can be used to support product updates, as it behaves the same under normal code execution conditions.

C1.11.1 Theory of operation

There are three types of register:

- a general control register FP_CTRL
- a Remap address register FP_REMAP
- FlashPatch comparator registers.

Separate comparators are used for instruction address comparison and literal address comparison.

The FlashPatch Control Register provides a global enable bit for the FPB, along with ID fields indicating the numbers of Code comparison and literal comparison registers provided.

The FlashPatch Remap Register is used to program the base address for the remap vectors. Comparator N will remap to address Remap_Base + 4N when configured for remapping and a match occurs. Bit [29] can be used to determine if remapping (instruction or data) is supported, see *FlashPatch Remap Register (FP_REMAP)* on page C1-64.

The instruction-matching FlashPatch Comparator Registers can be configured to remap the instruction or generate a breakpoint. The literal-matching comparators have fixed functionality, only supporting the remapping feature on data read accesses. Each comparator has its own enable bit which comes into effect when the global enable bit is set.

All comparators match word-aligned addresses (mask out address bits [1:0]) within the Code memory region (1st GB of the memory map), and only operate on read accesses. Data writes never match and will go to the original location.

Instruction matching where remapping is configured will always compare a word and remap a word for instruction issue. When instruction matching is configured to generate a breakpoint event, address matching is performed on the upper halfword, lower halfword or both halfwords. A halfword match generates a breakpoint event for a 16-bit instruction, and where the halfword match is on at least the first halfword of a 32-bit instruction. A halfword match on only the second halfword of a 32-bit instruction might generate a breakpoint event.

Note

It is IMPLEMENTATION DEFINED whether breakpoint events are generated when debug is disabled (DHCSR.C_DEBUGEN == 0 and DEMCR.MON_EN == 0). When the breakpoint is not generated, the breakpointed instruction exhibits its normal architectural behavior.

Literal matching on reads can be on a word, halfword or byte quantum of data. Matches will fetch the appropriate data from the remapped location.

The following restrictions apply:

- Unaligned literal accesses affected by remapping are IMPLEMENTATION DEFINED.
- Where an MPU is enabled, the MPU checking is performed on the original address, and the attributes applied to the remapped location. The remapped address is not checked by the MPU.
- Load exclusive accesses can be remapped, however, it is UNPREDICTABLE whether they are performed as exclusive accesses or not.
- Instruction matches on 32-bit instructions configured as a breakpoint must be configured to match the first halfword or both halfwords of the instruction. It is UNPREDICTABLE whether breakpoint matches on only the address of the second halfword of a 32-bit instruction generate a debug event.

C1.11.2 Register support for the FPB

The FPB register support is listed in Table C1-40:

Table C1-40 Flash Patch and Breakpoint register summary

Address	Type	Name	Reference
0xE0002000	R/W	FP_CTRL	FlashPatch control
0xE0002004	R/W	FP_REMAP	FlashPatch remapping address
0xE0002008	R/W	FP_COMP0	FlashPatch comparator 0 control
…		…	Number of comparators defined in FP_CTRL

Table C1-40 Flash Patch and Breakpoint register summary (continued)

Address	Type	Name	Reference
`0xE000223C`	R/W	FP_COMP141	where NUM_CODE = 127 and NUM_LIT = 15
`0xE0002FB4`[a]	RO		See Appendix B *ARMv7-M infrastructure IDs*, Table B-2 on page AppxB-3
`0xE0002FD0`	RO		ID space: See Appendix B *ARMv7-M infrastructure IDs* for more information.
…			
`0xE0002FFC`			

[a] CoreSight compliance for the lock status register is required.

See below for details on the FPB support registers.

FlashPatch Control Register (FP_CTRL)

Table C1-41 FP_CTRL

Bits	R/W	Name	Function
[31:15]	-	-	Reserved
[14:12]	RO	NUM_CODE2	Number of code comparators - most significant bits NUM_CODE = NUM_CODE2:NUM_CODE1
[11:8]	RO	NUM_LIT	The number of literal comparators supported. Literal comparators start at: FP_COMP[NUM_CODE] Last literal comp: FP_COMP[NUM_CODE+NUM_LIT-1] NUM_LIT == 0: no literal comparator support
[7:4]	RO	NUM_CODE1	Number of code comparators - least significant bits Code comparators start at: FP_COMP0 Last code comp: FP_COMP[NUM_CODE-1] NUM_CODE == 0: no code comparator support
[1]	R/W	KEY	RAZ on reads, MBO (must-be-one) for writes, otherwise the write to the register is ignored.
[0]	R/W	ENABLE	Enables the Flash Patch unit when set. This bit is cleared on a power-up reset.

FlashPatch Remap Register (FP_REMAP)

Table C1-42 FP_REMAP

Bits	R/W	Name	Function
[31:30]	RO		RAZ
[29]	RO		0: Remapping not supported 1: Hard-wired remap to SRAM region
[28:5]	R/W	REMAP	Remap base address in SRAM. The remap base address must be naturally aligned with respect to the number of words required to support (NUM_CODE+NUM_LIT-1) comparators, with a minimum 8-word alignment boundary. The field is UNKNOWN on reset and is RAZ/WI if only breakpoint functionality is supported.
[4:0]	-	-	Reserved. RAZ/WI

FlashPatch Comparator Register – instruction comparison (FP_COMPx)

Instruction comparators can be configured to remap the instruction to SRAM or to behave as a breakpoint.

Table C1-43 FP_COMPx instruction comparison

Bits	R/W	Name	Function
[31:30]	R/W	REPLACE	Defines the behavior when the COMP address is matched: 00[a]: remap to remap address. See FP_REMAP register. 01: breakpoint on lower halfword, upper is unaffected 10: breakpoint on upper halfword, lower is unaffected 11: breakpoint on both lower and upper halfwords. The field is UNKNOWN on reset.
[29]	-	-	Reserved.
[28:2]	R/W	COMP	Address to compare against within the Code segment of memory. The address for comparison is ’000’:COMP:’00’. The field is UNKNOWN on reset.
[1]	-	-	Reserved.
[0]	R/W	ENABLE	Comparator is enabled when this bit is set. This bit is cleared on a power-up reset. **Note** The master enable in FP_CTRL must also be set to enable a comparator.

a. If remapping is not supported, FP_CTRL.ENABLE == 1, and FP_COMPx.ENABLE bit == 1, the behavior associated with the value 0b00 is UNPREDICTABLE.

FlashPatch Comparator Register – literal comparison (FP_COMPx)

Literal comparators can only be configured with a remapping capability, see *FlashPatch Remap Register (FP_REMAP)* on page C1-64.

Table C1-44 FP_COMPx literal comparison

Bits	R/W	Name	Function
[31:30]	RO		RAZ/WI
[29]	-	-	Reserved.
[28:2]	R/W	COMP	Variable address field to compare against within the Code segment of memory. The field is UNKNOWN on reset. The address for comparison is ’000’:COMP:’00’.
[1]	-	-	Reserved.
[0]	R/W	ENABLE	Comparator is enabled when this bit is set. This bit is cleared on a power-up reset. **Note** The master enable in FP_CTRL must also be set to enable a comparator.

Part D

Appendices

Appendix A
CPUID

The CPUID scheme used on ARMv7-M aligns with the revised format ARM Architecture CPUID scheme. An architecture variant of `0xF` specified in the Main ID Register (see *System ID register support in the SCS* on page B3-10) indicates the revised format is being used. All ID registers are privileged access only. Privileged writes are ignored, and unprivileged data accesses cause a BusFault error. The appendix is made up of the following sections:

A.1 Core Feature ID Registers

The Core Feature ID registers are decoded in the System Control Space as defined in Table A-1.

Table A-1 Core Feature ID register support in the SCS

Address	Type	Reset Value	Function
`0xE000ED00`	Read Only	IMPLEMENTATION DEFINED	CPUID Base Register
`0xE000ED40`	Read Only	IMPLEMENTATION DEFINED	PFR0: Processor Feature register0
`0xE000ED44`	Read Only	IMPLEMENTATION DEFINED	PFR1: Processor Feature register1
`0xE000ED48`	Read Only	IMPLEMENTATION DEFINED	DFR0: Debug Feature register0
`0xE000ED4C`	Read Only	IMPLEMENTATION DEFINED	AFR0: Auxiliary Feature register0
`0xE000ED50`	Read Only	IMPLEMENTATION DEFINED	MMFR0: Memory Model Feature register0
`0xE000ED54`	Read Only	IMPLEMENTATION DEFINED	MMFR1: Memory Model Feature register1
`0xE000ED58`	Read Only	IMPLEMENTATION DEFINED	MMFR2: Memory Model Feature register2
`0xE000ED5C`	Read Only	IMPLEMENTATION DEFINED	MMFR3: Memory Model Feature register3
`0xE000ED60`	Read Only	IMPLEMENTATION DEFINED	ISAR0: ISA Feature register0
`0xE000ED64`	Read Only	IMPLEMENTATION DEFINED	ISAR1: ISA Feature register1
`0xE000ED68`	Read Only	IMPLEMENTATION DEFINED	ISAR2: ISA Feature register2
`0xE000ED6C`	Read Only	IMPLEMENTATION DEFINED	ISAR3: ISA Feature register3
`0xE000ED70`	Read Only	IMPLEMENTATION DEFINED	ISAR4: ISA Feature register4

Table A-1 Core Feature ID register support in the SCS (continued)

Address	Type	Reset Value	Function
0xE000ED74	Read Only	IMPLEMENTATION DEFINED	ISAR5: RESERVED (RAZ)
0xE000ED78	Read Only	IMPLEMENTATION DEFINED	RESERVED (RAZ)
0xE000ED7C	Read Only	IMPLEMENTATION DEFINED	RESERVED (RAZ)

Two values of the version fields have special meanings:

Field[ ] == all 0's the feature does not exist in this device, or the field is not allocated.

Field[ ] == all 1's the field has overflowed, and is now defined elsewhere in the ID space.

Where attribute fields or specific values apply only to other ARM Architecture profiles, they are marked as N/A (not applicable). All N/A values should be ignored, and the associated feature considered as not present.

All RESERVED fields in the Core Feature ID registers Read-as-Zero (RAZ).

A.1.1 Convention for CPUID attribute descriptions

The following convention is adopted specific to this appendix:

ARMv7 RESERVED The attribute is allocated within the ARM architecture, but does not apply to ARMv7-M. The attribute definition has been suppressed to minimize confusion.

RESERVED The attribute is not currently allocated within the ARM architecture.

N/A Not Applicable. The specific value of this attribute has been assigned within the ARM architecture but does not apply to ARMv7-M. The attribute value definition has been suppressed to minimize confusion.

A.2 Processor Feature register0 (ID_PFR0)

ID_PFR0[3:0] = State0 (T-bit == 0)

- 0 == no ARM encoding
- 1 == N/A

ID_PFR0[7:4] = State1 (T-bit == 1)

- 0 == No support for the Thumb instruction set
- 1 == Support for Thumb encoding before the introduction of Thumb-2 technology:
  - all instructions are 16-bit
  - a `BL` or `BLX` instruction is a pair of 16-bit instructions
  - 32-bit instructions other than `BL` and `BLX` cannot be encoded.
- 2 == Support for Thumb encoding after the introduction of Thumb-2 technology with `B` and `BL` 32-bit instructions and all 16-bit basic Thumb instructions.
- 3 == Support for Thumb encoding after the introduction of Thumb-2 technology with all basic 16-bit and 32-bit instructions.

ID_PFR0[15:8] = ARMv7 RESERVED

ID_PFR0[31:16] = RESERVED

A.3 Processor Feature register1 (ID_PFR1)

ID_PFR1[7:0] = ARMv7 RESERVED

ID_PFR1[11:8] = Microcontroller programmers' model

- 0 == not supported
- 1 == reserved
- 2 == two-stack support

ID_PFR1[31:12] = RESERVED

A.4 Debug Features register0 (ID_DFR0)

This register provides a high level view of the debug system. Further details are provided in the debug infrastructure itself.

ID_DFR0[19:0] = ARMv7 RESERVED

ID_DFR0[23:20] = Microcontroller Debug Model – memory mapped

- 0 == not supported
- 1 == Microcontroller debug v1 (ITMv1, DWTv1, optional ETM)

ID_DFR0[31:24] = RESERVED

A.5 Auxiliary Features register0 (ID_AFR0)

This register provides some freedom for IMPLEMENTATION DEFINED features to be registered against the CPUID. The field definitions are SUBJECT TO CHANGE, particularly across architecture licensees. Features can migrate over time if they get absorbed into the main architecture.

ID_AFR0[3:0] = IMPLEMENTATION DEFINED (architecture licensee)

- 0 == not supported / not in use

ID_AFR0[7:4] = IMPLEMENTATION DEFINED (architecture licensee)

- 0 == not supported / not in use

ID_AFR0[11:8] = IMPLEMENTATION DEFINED (architecture licensee)

- 0 == not supported / not in use

ID_AFR0[15:12] = IMPLEMENTATION DEFINED (architecture licensee)

- 0 == not supported / not in use

ID_AFR0[31:16] = RESERVED

A.6 Memory Model Feature registers

Four registers contain general information on the memory model and memory management support.

A.6.1 Memory Model Feature register0 (ID_MMFR0)

ID_MMFR0[3:0] = ARMv7 RESERVED

ID_MMFR0[7:4] = PMSA support

- 0 == not supported
- 1 == IMPLEMENTATION DEFINED (N/A)
- 2 == PMSA base (features as defined for ARMv6) (N/A)
- 3 == PMSAv7 (base plus subregion support)

ID_MMFR0[11:8] = cache coherence support

- 0 == no shared support
- 1 == partial-inner-shared coherency (coherency amongst some - but not all - of the entities within an inner-coherent domain)
- 2 == full-inner-shared coherency (coherency amongst all of the entities within an inner-coherent domain)
- 3 == full coherency (coherency amongst all of the entities)

The values 2 and 3 can only be distinguished when Outer non-sharable is supported

ID_MMFR0[15:12] = Outer non-sharable support

- 0 == Outer non-sharable not supported
- 1 == Outer sharable supported

ID_MMFR0[19:16] = ARMv7 RESERVED

ID_MMFR0[23:20] = Auxiliary register support

- 0 == not supported
- 1 == Auxiliary control register

ID_MMFR0[27:24] = ARMv7 RESERVED

ID_MMFR0[31:28] = RESERVED

A.6.2 Memory Model Feature register1 (ID_MMFR1)

ID_MMFR1[31:0] = ARMv7 RESERVED

A.6.3 Memory Model Feature register2 (ID_MMFR2)

ID_MMFR2[23:0] = ARMv7 RESERVED

ID_MMFR2[27:24] = wait for interrupt stalling

- 0 == not supported
- 1 == wait for interrupt supported

ID_MMFR2[31:28] = ARMv7 RESERVED

A.6.4 Memory Model Feature register3 (ID_MMFR3)

ID_MMFR3[11:0] = ARMv7 RESERVED

ID_MMFR3[31:12] = RESERVED

A.7 Instruction Set Attribute registers – background information

This section provides background information on the instructions which form the basic instruction set, and some rules about the allocation of instructions to the different attribute fields in the Instruction Set Attribute registers.

A.7.1 The basic instruction set

These instructions only depend on an "instruction encoding" attribute in the CPUID – i.e. if an instruction encoding is present, all basic instructions that have encodings in that instruction set *must* be present.

A.7.2 General rules

The rules about an instruction being basic do not guarantee that it is available in any particular instruction set - for instance, `MOV R0,#123456789` is a basic instruction by the rules below, but is not available in any ARM instruction sets to date.

Being conditional or unconditional never makes any difference to whether an instruction is a basic instruction.

A.7.3 Q flag support

The Q flag is present in the APSR when:

(MultS_instrs 2) OR (Saturate_instrs 1) OR (SIMD_instrs 1)

Note

This value of MultS_instrs and the Saturate_instrs attribute are ARMv7 RESERVED.

A.7.4 MOV instructions

These are in the basic instruction set if the source operand is an immediate or an unshifted register.

If their second operand is a shifted register, treat them as instead being an `ASR`, `LSL`, `LSR`, `ROR` or "`RRX`" instruction, as described in the following section.

A.7.5 Non-MOV data-processing instructions

These are:

```
ADC   ADD   AND   ASR   BIC   CMN   CMP   EOR   LSL   LSR   MVN
NEG   ORN   ORR   ROR   RRX   RSB   RSC   SBC   SUB   TEQ   TST
```

These instructions are in the basic instruction set for ARMv7-M if the second (or only for `MVN`) source operand is an immediate or an unshifted register.

If this condition is false, they are non-basic instructions, controlled by the PSR_instrs attribute and/or the WithShifts_instrs attribute.

A.7.6 Multiply instructions

`MUL` instructions are always basic; all other multiply instructions and all multiply-accumulate instructions are non-basic.

A.7.7 Branches

All `B` and `BL` instructions are basic instructions.

A.7.8 Load/store single instructions

These are:

```
  LDR   LDRB   LDRH   LDRSB   LDRSH   STR   STRB   STRH
```

These instructions are in the basic instruction set if the addressing mode is of one of the following forms:

```
[Rn, #immediate]
[Rn, #-immediate]
[Rn, Rm]
[Rn, -Rm]
```

A load/store single instruction with any other addressing mode is under the control of one or more of the attributes WithShifts_instrs, Writeback_instrs or Unpriv_instrs.

A.7.9 Load/store multiple instructions

These are:

```
  LDM<mode>   STM<mode>   PUSH   POP
```

Where <mode> is:

```
IA Rn,
IA Rn!,
DB Rn,  DB Rn!,
```

or their corresponding FD/EA synonyms.

They are basic because they are fundamental to good code generation. In particular, `PUSH` and `POP` both have the implied addressing mode FD R13! and are essential for good procedure prologues and epilogues. The other addressing modes listed can make a considerable difference to the code density of structure copy, load and store, and also to their performance on low-end implementations.

A.8 Instruction Set Attribute registers – details

Six registers are currently allocated for instruction set attributes.

A.8.1 Instruction Set Attributes Register0 (ID_ISAR0)

ID_ISAR0[3:0] = ARMv7 RESERVED

ID_ISAR0[7:4] = BitCount instructions

ID_ISAR0[11:8] = BitField instructions

ID_ISAR0[15:12] = CmpBranch instructions

ID_ISAR0[19:16] = Coprocessor instructions

ID_ISAR0[23:20] = Debug instructions

ID_ISAR0[27:24] = Divide instructions

ID_ISAR0[31:28] = reserved

BitCount_instrs:

- 0 if no bit-counting instructions present
- 1 adds `CLZ`

Bitfield_instrs:

- 0 if no bitfield instructions present
- 1 adds `BFC`, `BFI`, `SBFX`, `UBFX`

CmpBranch_instrs:

- 0 if no combined compare-and-branch instructions present
- 1 adds `CB{N}Z`

Coproc_instrs:

- 0 if no coprocessor support, other than for separately attributed architectures such as CP15 or VFP
- 1 adds generic `CDP`, `LDC`, `MCR`, `MRC`, `STC`
- 2 adds generic `CDP2`, `LDC2`, `MCR2`, `MRC2`, `STC2`
- 3 adds generic `MCRR`, `MRRC`,
- 4 adds generic `MCRR2`, `MRRC2`

Debug_instrs:

- 0 if no debug instructions present
- 1 adds `BKPT`

Divide_instrs:

- 0 if no divide instructions present
- 1 adds `SDIV`, `UDIV` (v1 – quotient only result)

A.8.2 Instruction Set Attributes Register1 (ID_ISAR1)

ID_ISAR1[11:0] = ARMv7 RESERVED

ID_ISAR1[15:12] = Extend instructions

ID_ISAR1[19:16] = IfThen instructions

ID_ISAR1[23:20] = Immediate instructions

ID_ISAR1[27:24] = Interwork instructions

ID_ISAR1[31:28] = ARMv7 RESERVED

Extend_instrs:

0 if no scalar (i.e. non-SIMD) sign/zero-extend instructions present

- 1 adds `SXTB`, `SXTH`, `UXTB`, `UXTH`
- 2 == N/A

Note

The shift options on these instructions are also controlled by the WithShifts_instrs attribute.

IfThen_instrs (T):

- 0 if IT instructions not present
- 1 adds IT instructions (and IT bits in PSRs)

Immediate_instrs:

- 0 if no special immediate-generating instructions present
- 1 adds `ADDW`, `MOVW`, `MOVT`, `SUBW`

Interwork_instrs:

- 0 if no interworking instructions supported
- 1 adds `BX` (and T bit in PSRs)
- 2 adds `BLX`, and PC loads have `BX`-like behavior
- 3 == N/A

A.8.3 Instruction Set Attributes Register2 (ID_ISAR2)

ID_ISAR2[3:0] = LoadStore instructions

ID_ISAR2[7:4] = MemoryHint instructions

ID_ISAR2[11:8] = Multi-Access interruptible instructions

ID_ISAR2[15:12] = Multiply instructions

ID_ISAR2[19:16] = Multiply instructions (advanced, signed)

ID_ISAR2[23:20] = Multiply instructions (advanced, unsigned)

ID_ISAR2[27:24] = ARMv7 RESERVED

ID_ISAR2[31:28] = Reversal instructions

LoadStore_instrs:

- 0 if no additional normal load/store instructions present
- 1 adds `LDRD`/`STRD`

MemHint_instrs:

- 0 if no memory hint instructions present
- 1 adds `PLD`
- 2 adds `PLD` (ie a repeat on value 1)
- 3 adds `PLI`

MultiAccessInt_instrs:

- 0 if the (`LDM`/`STM`) instructions are non-interruptible
- 1 if the (`LDM`/`STM`) instructions are restartable
- 2 if the (`LDM`/`STM`) instructions are continuable

Mult_instrs:

- 0 if only `MUL` present
- 1 adds `MLA`
- 2 adds `MLS`

MultS_instrs:

- 0 if no signed multiply instructions present
- 1 adds SMULL, SMLAL
- 2 == N/A
- 3 == N/A

MultU_instrs:

- 0 if no unsigned multiply instructions present
- 1 adds `UMULL`, `UMLAL`
- 2 == N/A

Reversal_instrs:

- 0 if no reversal instructions present
- 1 adds `REV`, `REV16`, `REVSH`
- 2 adds `RBIT`

A.8.4 Instruction Set Attributes Register3 (ID_ISAR3)

ID_ISAR3[3:0] = Saturate instructions

ID_ISAR3[7:4] = SIMD instructions

ID_ISAR3[11:8] = SVC instructions

ID_ISAR3[15:12] = SyncPrim instructions

ID_ISAR3[19:16] = TableBranch instructions

ID_ISAR3[23:20] = ThumbCopy instructions

ID_ISAR3[27:24] = TrueNOP instructions

ID_ISAR3[31:28] = ARMv7 RESERVED

Saturate_instrs:

- 0 if no non-SIMD saturate instructions present
- 1 == N/A

SIMD_instrs:

- 0 if no SIMD instructions present
- 1 adds `SSAT`, `USAT` (and the Q flag in the PSRs)
- 2 reserved
- 3 == N/A

SVC_instrs:

- 0 if no SVC (SWI) instructions present
- 1 adds SVC (SWI)

SyncPrim_instrs:

- 0 if no synchronization primitives present
- 1 adds `LDREX`, `STREX`
- 2 adds `LDREXB`, `LDREXH`, `LDREXD`, `STREXB`, `STREXH`, `STREXD`, `CLREX`(N/A)

Note

No `LDREXD` or `STREXD` in ARMv7-M. This attribute is used in conjunction with the SyncPrim_instrs_frac attribute in ID_ISAR4[23:20].

TabBranch_instrs (T):

- 0 if no table-branch instructions present
- 1 adds `TBB`, `TBH`

ThumbCopy_instrs (T):

- 0 if Thumb `MOV`(register) instruction does not allow low reg -> low reg
- 1 adds Thumb `MOV`(register) low reg -> low reg and the `CPY` alias

TrueNOP_instrs

- 0 if "true NOP" instructions not present - that is, NOP instructions with no register dependencies
- 1 adds "true NOP”, and the capability of additional “NOP compatible hints”

A.8.5 Instruction Set Attributes Register~~3~~ (ID_ISAR4)

ID_ISAR4[3:0] = Unprivileged instructions

ID_ISAR4[7:4] = WithShift instructions

ID_ISAR4[11:8] = Writeback instructions

ID_ISAR4[15:12] = ARMv7 RESERVED

ID_ISAR4[19:16] = Barrier instructions

ID_ISAR4[23:20] = SyncPrim_instrs_frac

ID_ISAR4[27:24] = PSR_M_instrs

ID_ISAR4[31:28] = RESERVED

Unpriv_instrs:

- 0 if no "T variant" instructions exist
- 1 adds LDRBT, LDRT, STRBT, STRT
- 2 adds LDRHT, LDRSBT, LDRSHT, STRHT

WithShifts_instrs:

- 0 if non-zero shifts only support MOV and shift instructions (see notes)
- 1 shifts of loads/stores over the range LSL 0-3
- 2 reserved
- 3 adds other constant shift options.
- 4 adds register-controlled shift options.

Note

All additions only apply in cases where the encoding supports them - e.g. there is no difference between levels 3 and 4 in the Thumb instruction set.

MOV instructions with shift options are treated as ASR, LSL, LSR, ROR or RRX instructions, as described in *Non-MOV data-processing instructions* on page AppxA-10.

Writeback_instrs:

- 0 if only non-writeback addressing modes present, except that LDMIA/STMDB/PUSH/POP instructions support writeback addressing.

- 1 adds all currently-defined writeback addressing modes in ARMv7.

Barrier_instrs

- 0 if no barrier instructions supported
- 1 adds `DMB`, `DSB`, `ISB` barrier instructions

Syncprim_instrs_frac:

- When SyncPrim_instrs = 1
  - — 0 if no additional support
  - — 1 reserved
  - — 2 reserved
  - — 3 adds `CLREX`, `LDREXB`, `STREXB`, `LDREXH`, `STREXH`

PSR_M_instrs

- 0 if instructions not present
- 1 adds `CPS`, `MRS`, and `MSR` instructions (M-profile forms)

Appendix B
ARMv7-M infrastructure IDs

ARMv7-M implementations support SCS, FPB, DWT and ITM blocks along with a ROM table as illustrated in Table C1-3 on page C1-4. The CoreSight architecture programmers' model is defined in [3] with each 4KB register space subdivided into four sections:

- A component ID (offset `0xFF0` to `0xFFF`)
- A peripheral ID (offset `0xFD0` to `0xFEF`)
- CoreSight management registers (offset `0xF00` to `0xFCF`)
- Device specific registers (offset `0x000` to `0xEFF`)

For ARMv7-M, the component ID registers are required for the ROM table, and the CoreSight management lock access mechanism is defined for the DWT, ITM, FPB and TPIU blocks. Otherwise all ID and management registers are reserved, with the recommendation that they are CoreSight compliant or RAZ to encourage commonality of support across debug toolchains.

Note

The lock mechanism only applies to software access from the core to the affected block. DAP access is always allowed, meaning the lock status register must RAZ from the DAP.

To determine the topology of the ARMv7-M debug infrastructure, ROM table entries indicate whether a block is present. Presence of a block guarantees support of the ARMv7-M programming requirements for DWT, ITM, FPB and TPIU. Additional functionality requires additional support, where CoreSight is the recommended framework.

The CPUID support in the SCS should be used to determine details of the architecture variant and features supported by the core.

The component ID and peripheral ID register general formats are as shown in Table B-1.

Table B-1 Component and Peripheral ID register formats

Address Offset	Value	Symbol	Name	Reference
`0xFFC`	`0x000000B1`	CID3	Component ID3	Preamble
`0xFF8`	`0x00000005`	CID2	Component ID2	Preamble
`0xFF4`	`0x000000X0`	CID1	Component ID1	bits [7:4] Component Class bits [3:0] Preamble
`0xFF0`	`0x0000000D`	CID0	Component ID0	Preamble
`0xFEC`	`0x000000YY`	PID3	Peripheral ID3	bits [7:4] RevAnd - minor revision field bits [3:0] non-zero == Customer modified block
`0xFE8`	`0x000000YX`	PID2	Peripheral ID2	bits [7:4] Revision bit [3] == 1: JEDEC assigned ID fields bits [2:0] JEP106 ID code [6:4]
`0xFE4`	`0x000000XY`	PID1	Peripheral ID1	bits [7:4] JEP106 ID code [3:0] bits [3:0] Part Number [11:8]
`0xFE0`	`0x000000YY`	PID0	Peripheral ID0	Part Number [7:0]
`0xFDC`	`0x00000000`	PID7	Peripheral ID7	Reserved
`0xFD8`	`0x00000000`	PID6	Peripheral ID6	Reserved
`0xFD4`	`0x00000000`	PID5	Peripheral ID5	Reserved
`0xFD0`	`0x000000YX`	PID4	Peripheral ID4	bits [7:4] 4KB count bits [3:0] JEP106 continuation code

1. For ARMv7-M, all CoreSight registers are accessed as words. Any 8-bit or 16-bit registers defined in the *CoreSight Architecture Specification* are accessed as zero-extended words.
2. For the Value column:
   - X: CoreSight architected values (fixed bit field, component class, JEDEC assigned).
   - Y: IMPLEMENTATION DEFINED
3. The JEDEC defined fields refer to the block designer’s JEDEC code. The combination of part number, designer and component class fields must be unique.
4. For more details on the bit fields, see the CoreSight programmers’ model in the *CoreSight Architecture Specification*.

The CoreSight management lock mechanism registers are summarized in Table B-2.

Table B-2 ARMv7-M and CoreSight management registers

Component Class[a]	Address Offset	Type	Register Name	Notes
	`0xFB8-0xFCF`		reserved	
`0x9`	`0xFB4`	RO	Lock Status (LSR)	optional in ARMv7-M, or RAZ
`0x9`	`0xFB0`	WO	Lock Access (LAR)	optional in ARMv7-M, reads UNKNOWN
	`0xF00-0xFAF`		reserved	
	ARM recommends that all reserved space is CoreSight compliant or RAZ. See the programmers' model in the *CoreSight Architecture Specification* for a complete CoreSight management register list and register format details.			

[a] For information on component classes, see the Component ID register information in the *CoreSight Architecture Specification*.

Appendix C
Legacy Instruction Mnemonics

This appendix provides information about the Unified Assembler Language equivalents of older assembler language instruction mnemonics.

It contains the following sections:

C.1 Thumb instruction mnemonics

The following table shows the pre-UAL assembly syntax used for Thumb instructions before the introduction of Thumb-2 technology and the equivalent UAL syntax for each instruction. It can be used to translate correctly-assembling pre-UAL Thumb assembler code into UAL assembler code.

This table is not intended to be used for the reverse translation from UAL assembler code to pre-UAL Thumb assembler code.

In this table, 3-operand forms of the equivalent UAL syntax are used, except in one case where a 2-operand form needs to be used to ensure that the same instruction encoding is selected by a UAL assembler as was selected by a pre-UAL Thumb assembler.

Table C-1 Pre-UAL assembly syntax

Pre-UAL Thumb syntax	Equivalent UAL syntax	Notes
ADC <Rd>, <Rm>	ADCS <Rd>, <Rd>, <Rm>	
ADD <Rd>, <Rn>, #<imm>	ADDS <Rd>, <Rn>, #<imm>	
ADD <Rd>, #<imm>	ADDS <Rd>, #<imm>	
ADD <Rd>, <Rn>, <Rm>	ADDS <Rd>, <Rn>, <Rm>	
ADD <Rd>, SP	ADD <Rd>, SP, <Rd>	
ADD <Rd>, <Rm>	ADDS <Rd>, ~~<Rd>,~~ <Rm> ADD <Rd>, <Rd>, <Rm>	If <Rd> and <Rm> are both R0-R7, otherwise (and <Rm> is not SP)
ADD <Rd>, PC, #<imm> ADR <Rd>, <label>	ADD <Rd>, PC, #<imm> ADR <Rd>, <label>	ADR form preferred where possible
ADD <Rd>, SP, #<imm>	ADD <Rd>, SP, #<imm>	
ADD SP, #<imm>	ADD SP, SP, #<imm>	
AND <Rd>, <Rm>	ANDS <Rd>, <Rd>, <Rm>	
ASR <Rd>, <Rm>, #<imm>	ASRS <Rd>, <Rm>, #<imm>	
ASR <Rd>, <Rs>	ASRS <Rd>, <Rd>, <Rs>	
B<cond> <label>	B<cond> <label>	
B <label>	B <label>	
BIC <Rd>, <Rm>	BICS <Rd>, <Rd>, <Rm>	
BKPT <imm>	BKPT <imm>	

Table C-1 Pre-UAL assembly syntax (continued)

Pre-UAL Thumb syntax	Equivalent UAL syntax	Notes
BL <label>	BL <label>	
BLX <Rm>	BLX <Rm>	<Rm> can be a high register
BX <Rm>	BX <Rm>	<Rm> can be a high register
CMN <Rn>, <Rm>	CMN <Rn>, <Rm>	
CMP <Rn>, #<imm>	CMP <Rn>, #<imm>	
CMP <Rn>, <Rm>	CMP <Rn>, <Rm>	<Rd> and <Rm> can be high registers.
CPS<effect> <iflags>	CPS<effect> <iflags>	
CPY <Rd>, <Rm>	MOV <Rd>, <Rm>	
EOR <Rd>, <Rm>	EORS <Rd>, <Rd>, <Rm>	
LDMIA <Rn>!, <registers>	LDMIA <Rn>, <registers> LDMIA <Rn>!, <registers>	If <Rn> listed in <registers>, otherwise
LDR <Rd>, [<Rn>, #<imm>]	LDR <Rd>, [<Rn>, #<imm>]	<Rn> can be SP
LDR <Rd>, [<Rn>, <Rm>]	LDR <Rd>, [<Rn>, <Rm>]	
LDR <Rd>, [PC, #<imm>] LDR <Rd>, <label>	LDR <Rd>, [PC, #<imm>] LDR <Rd>, <label>	<label> form preferred where possible
LDRB <Rd>, [<Rn>, #<imm>]	LDRB <Rd>, [<Rn>, #<imm>]	
LDRB <Rd>, [<Rn>, <Rm>]	LDRB <Rd>, [<Rn>, <Rm>]	
LDRH <Rd>, [<Rn>, #<imm>]	LDRH <Rd>, [<Rn>, #<imm>]	
LDRH <Rd>, [<Rn>, <Rm>]	LDRH <Rd>, [<Rn>, <Rm>]	
LDRSB <Rd>, [<Rn>, <Rm>]	LDRSB <Rd>, [<Rn>, <Rm>]	
LDRSH <Rd>, [<Rn>, <Rm>]	LDRSH <Rd>, [<Rn>, <Rm>]	
LSL <Rd>, <Rm>, #<imm>	MOVS <Rd>, <Rm> LSLS <Rd>, <Rm>, #<imm>	If <imm> == 0, otherwise
LSL <Rd>, <Rs>	LSLS <Rd>, <Rd>, <Rs>	
LSR <Rd>, <Rm>, #<imm>	LSRS <Rd>, <Rm>, #<imm>	
LSR <Rd>, <Rs>	LSRS <Rd>, <Rd>, <Rs>	

Table C-1 Pre-UAL assembly syntax (continued)

Pre-UAL Thumb syntax	Equivalent UAL syntax	Notes
MOV <Rd>, #<imm>	MOVS <Rd>, #<imm>	
MOV <Rd>, <Rm>	ADDS <Rd>, <Rm>, #0 MOV <Rd>, <Rm>	If <Rd> and <Rm> are both R0-R7, otherwise
MUL <Rd>, <Rm>	MULS <Rd>, <Rm>, <Rd>	
MVN <Rd>, <Rm>	MVNS <Rd>, <Rm>	
NEG <Rd>, <Rm>	RSBS <Rd>, <Rm>, #0	
ORR <Rd>, <Rm>	ORRS <Rd>, <Rd>, <Rm>	
POP <registers>	POP <registers>	<registers> can include PC
PUSH <registers>	PUSH <registers>	<registers> can include LR
REV <Rd>, <Rn>	REV <Rd>, <Rn	
REV16 <Rd>, <Rn>	REV16 <Rd>, <Rn	
REVSH <Rd>, <Rn>	REVSH <Rd>, <Rn	
ROR <Rd>, <Rs>	RORS <Rd>, <Rd>, <Rs>	
SBC <Rd>, <Rm>	SBCS <Rd>, <Rd>, <Rm>	
STMIA <Rn>!, <registers>	STMIA <Rn>!, <registers>	
STR <Rd>, [<Rn>, #<imm>]	STR <Rd>, [<Rn>, #<imm>]	<Rn> can be SP
STR <Rd>, [<Rn>, <Rm>]	STR <Rd>, [<Rn>, <Rm>]	
STRB <Rd>, [<Rn>, #<imm>]	STRB <Rd>, [<Rn>, #<imm>]	
STRB <Rd>, [<Rn>, <Rm>]	STRB <Rd>, [<Rn>, <Rm>]	
STRH <Rd>, [<Rn>, #<imm>]	STRH <Rd>, [<Rn>, #<imm>]	
STRH <Rd>, [<Rn>, <Rm>]	STRH <Rd>, [<Rn>, <Rm>]	
SUB <Rd>, <Rn>, #<imm>	SUBS <Rd>, <Rn>, #<imm>	
SUB <Rd>, #<imm>	SUBS <Rd>, #<imm>	
SUB <Rd>, <Rn>, <Rm>	SUBS <Rd>, <Rn>, <Rm>	
SUB SP, #<imm>	SUB SP, SP, #<imm>	

Table C-1 Pre-UAL assembly syntax (continued)

Pre-UAL Thumb syntax	Equivalent UAL syntax	Notes
SWI <imm>	SVC <imm>	
SXTB <Rd>, <Rm>	SXTB <Rd>, <Rm>	
SXTH <Rd>, <Rm>	SXTH <Rd>, <Rm>	
TST <Rn>, <Rm>	TST <Rn>, <Rm>	
UXTB <Rd>, <Rm>	UXTB <Rd>, <Rm>	
UXTH <Rd>, <Rm>	UXTH <Rd>, <Rm>	

C.2 Pre-UAL pseudo-instruction NOP

In pre-UAL assembler code, `NOP` is a pseudo-instruction, equivalent to `MOV R8,R8` in Thumb code.

Assembling the `NOP` mnemonic as UAL will not change the functionality of the code, but will change:

- the instruction encoding selected
- the architecture variants on which the resulting binary will execute successfully, because the Thumb version of the `NOP` instruction was introduced in ARMv6T2.

To avoid the change in Thumb code, replace `NOP` in the assembler source code with `MOV R8,R8`, before assembling as UAL.

Note

The pre-UAL pseudo-instruction is different for ARM code where it is equivalent to `MOV R0,R0`.

Appendix D
Deprecated Features in ARMv7-M

Some features of the Thumb instruction set are deprecated in ARMv7. Deprecated features affecting instructions supported by ARMv7-M are as follows:

- use of the PC as `<Rd>` or `<Rm>` in a 16-bit `ADD` (SP plus register) instruction
- use of the SP as `<Rm>` in a 16-bit `ADD` (SP plus register) instruction
- use of the SP as `<Rm>` in a 16-bit `CMP` (register) instruction
- use of `MOV` (register) instructions in which both `<Rd>` and `<Rm>` are the SP or PC.
- use of `<Rn>` as the lowest-numbered register in the register list of a 16-bit STM instruction with base register writeback

The following additional feature in ARMv7-M is deprecated:

- support of a 4-byte aligned stack (CCR.STKALIGN == ’0’).

Appendix E
Debug ITM and DWT packet protocol

The ITM emits packets to the TPIU when a stimulus register is written, or a timestamp issued.

The DWT emits packets when a data trace event triggers, PC sampling occurs, or one of the profile counters wraps. The packet protocol used is described in this appendix. This appendix consists of the following sections:

- *Packet Types* on page AppxE-2
- *DWT packet formats* on page AppxE-8

E.1 Packet Types

Diagnostic information is output in packets with a byte protocol. These packets are 1-5 bytes in size, comprising:

- one byte header
- 0-4 byte payload.

Synchronization packets are the exception, comprising a unique 6-byte sequence for bit & byte synchronization.

The header encodings are shown below:

Table E-1 ITM and DWT general packet formats

Description	Value	Payload	Category	Remarks
Sync	S0000000	None	Synchronization	{47{1'b0}} followed by 1'b1 (Matches ETM protocol)
Overflow	01110000	0	Protocol	Overflow

Time Stamp	CDDD0000	0 to 4 bytes	Protocol	D = data (!=000) – time
				C = continuation

Extension	CDDD1S00	0 to 4 bytes	Protocol	S = source (ITM / DWT)
				D = data – page info
				C = continuation

Reserved	Cxxx0100	0 to 4 bytes	Reserved	C = continuation

Instrumentation	AAAAA0SS	1, 2 or 4 bytes	Software Source (Application)	SS = size (!=00) of payload
				A = SW Source Address

Hardware Source	AAAAA1SS	1, 2 or 4 bytes	Hardware Source (Diagnostics)	SS = size (!=00) of payload
				A = HW packet type discriminator

Apart from synchronization, the maximum packet size is five bytes. A maximum of four continuation bits can be set within a packet. Packets with five or more continuation bits set are UNPREDICTABLE.

All packets are transmitted over a serial port least significant bit (LSB) first. Where bytes are illustrated in the following subsections, the figure convention is to indicate the most significant bit (MSB) to the left, and the LSB to the right.

bit [7] (MSB)	bit [6]	bit [5]	bit [4]	bit[3]	bit [2]	bit [1]	bit [0] (LSB)

E.1.1 Sync Packet

Sync packets are unique patterns in the bit-stream that allow capture hardware to identify bit-to-byte alignment. While Sync packets can be emitted on an asynchronous interface, they are only necessary on a synchronous port. See the *CoreSight Architecture Specification* for more information. A sync packet is at least forty-seven 1'b0 followed by one 1'b1. Any page event register (see *Software Instrumentation Packet* on page AppxE-5) is cleared when a sync packet is generated.

Table E-2 Sync packet (matches ETM format)

	bit [7]							bit [0]
byte 1	0	0	0	0	0	0	0	0
byte 2	0	0	0	0	0	0	0	0
byte 3	0	0	0	0	0	0	0	0
byte 4	0	0	0	0	0	0	0	0
byte 5	0	0	0	0	0	0	0	0
byte 6	1	0	0	0	0	0	0	0

E.1.2 Overflow Packet

Overflow packets are generated under the following conditions:

- a Stimulus Port register write when the Stimulus port output buffer is full
- timestamp counter overflow
- DWT (Hardware Source) packet generation when the DWT output buffer is full.

Table E-3 Overflow packet format

bit [7]							bit [0]
0	1	1	1	0	0	0	0

E.1.3 Timestamp Packet

Time stamp packets encode timestamp information, generic control and synchronization. The compression scheme uses delta timestamps, where the timestamp value represents the interval since the previous timestamp packet value was output. Each time a timestamp packet is output, the timestamp counter is cleared to zero.

Table E-4 Timestamp packet format 1

	bit [7]							bit [0]
byte 1	1	1	TC[1]	TC[0]	0	0	0	0
byte 2	C	TS[6]	TS[5]	TS[4]	TS[3]	TS[2]	TS[1]	TS[0]
byte 3	C	TS[13]	TS[12]	TS[11]	TS[10]	TS[9]	TS[8]	TS[7]
byte 4	C	TS[20]	TS[19]	TS[18]	TS[17]	TS[16]	TS[15]	TS[14]
byte 5	0	TS[27]	TS[26]	TS[25]	TS[24]	TS[23]	TS[22]	TS[21]

Packet format 1 is a multi-byte packet (identified by bit [7] == bit [6] == 1), with the TC[1:0] field interpreted as follows:

0 The timestamp value is emitted synchronously to ITM/DWT data. The value in the TS[] field is the timestamp counter value at the time the ITM/DWT packet(s) is/are generated.

1 The timestamp value emitted is delayed with respect to the event data. The value in the TS[] field is the timestamp counter value at the time the timestamp packet is generated.

Note

The timestamp value that correlates with the previous ITM/DWT packet is UNKNOWN, however, its value is within the range governed by the previous and current timestamp values.

2 The emission of the ITM/DWT packet that is associated with this timestamp packet was delayed with respect to the event. The value in the TS[] field is the timestamp counter value at the time the ITM/DWT packet(s) is/are generated.

This encoding indicates that the ITM/DWT packet was delayed with respect to other TPIU output packets.

3 The associated ITM/DWT packet is delayed with respect to the event and this timestamp packet is delayed with respect to the event packet. This is a combination of the events indicated by values 1 and 2.

Note

If the trace system is fully utilized by higher priority trace data, then timestamp packets might be lost. This can result in a block (or blocks) of ITM/DWT data that do not have any timestamp information available. Timing information is then uncertain until the next timestamp packet is output.

Table E-5 Timestamp packet format 2

	bit [7]							bit [0]
byte 1	0	TS[2]	TS[1]	TS[0]	0	0	0	0

Packet format 2 is a single byte packet (identified by bit [7] == 0). The format is an optimization and is equivalent to a format 1 packet containing TC[1:0] == 0 and a timestamp value in the range 1..6.

The TS[2:0] field is interpreted as follows:

0 See *Sync Packet* on page AppxE-3.

1 to 6 The timestamp value is emitted synchronously to ITM/DWT data. The value in the TS[] field is the timestamp counter value at the time the ITM/DWT packet(s) is/are generated.

7 See *Overflow Packet* on page AppxE-3.

E.1.4 Software Instrumentation Packet

Software instrumentation (event) packets are generated by writing to a stimulus port of the ITM.

Table E-6 Software instrumentation packet format

	bit [7]							bit [0]
byte 1	A[4]	A[3]	A[2]	A[1]	A[0]	0	S	S
byte 2	Payload[7:0]							
byte 3	Payload[15:8]							
byte 4	Payload[23:16]							
byte 5	Payload[31:24]							

SS = Payload Size in bits

01	8-bit payload, 2 byte packet
10	16-bit payload, 3 byte packet
11	32-bit payload, 5 byte packet
00	invalid

A[4:0] = the stimulus port number (0-31). A single-byte software extension packet indicates which page subsequent stimulus packets are applied to (1 of 8, supporting up to 256 stimulus ports).

Any page register implemented is cleared on receipt of a synchronization packet.

Payload[{31:0}, {15:0}, {7:0}] = Data written (the software event value) by the application

E.1.5 Hardware Source Packet

Hardware events are generated from the DWT block. For more details on the defined packet formats, see *DWT packet formats* on page AppxE-8.

Table E-7 Hardware source packet format

	bit [7]							bit [0]
byte 1	A[4]	A[3]	A[2]	A[1]	A[0]	1	S	S
byte 2	Payload[7:0]							
byte 3	Payload[15:8]							
byte 4	Payload[23:16]							
byte 5	Payload[31:24]							

SS = Payload Size in bits:

01	8-bit payload, 2 byte packet
10	16-bit payload, 3 byte packet
11	32-bit payload, 5 byte packet
00	invalid

Payload[{31:0}, {15:0}, {7:0}] = Data written from the hardware source

E.1.6 Extension Packets

Extension packets always follow the source. Extension packets are used to add additional sideband information to the source.

Table E-8 Extension packet format

	bit [7]							bit [0]
byte 1	C	EX[2]	EX[1]	EX[0]	1	SH	0	0
byte 2	C	EX[9]	EX[8]	EX[7]	EX[6]	EX[5]	EX[4]	EX[3]

Table E-8 Extension packet format (continued)

byte 3	C	EX[16]	EX[15]	EX[14]	EX[13]	EX[12]	EX[11]	EX[10]
byte 4	C	EX[23]	EX[22]	EX[21]	EX[20]	EX[19]	EX[18]	EX[17]
byte 5	EX[31]	EX[30]	EX[29]	EX[28]	EX[27]	EX[26]	EX[25]	EX[24]

C = continuation bit – a byte follows.

EX[N:0] = byte packing of extension information.

SH = software source(0), hardware source(1)

A single byte software extension packet is defined as providing a page register capability for stimulus ports. All other extension packets (C == 1, or SH == 1) are reserved.

E.1.7 Reserved Encodings

Table E-9 Reserved packet encodings

	bit [7]							bit [0]
byte 1	C	E2	E1	E0	0	1	0	0
byte 2	C	x	x	x	x	x	x	x
byte 3	C	x	x	x	x	x	x	x
byte 4	C	x	x	x	x	x	x	x
byte 5	x	x	x	x	x	x	x	x

E.1.8 Multiple Source Arbitration

Trace packets are generated from three sources – ITM derived software events, timestamps (TS) and DWT derived hardware events. When multiple sources are trying to emit data, arbitration is performed in accordance with the following priority assignments.

ITM (Highest)	Priority Level 0
DWT	Priority Level 1
TS (Lowest)	Priority Level 2

This ensures a quality of service for ITM output over hardware generated DWT events, and that timestamps are emitted once other source queues are empty.

E.2 DWT packet formats

DWT generated packets adopt the general Hardware Source packet format defined in *Hardware Source Packet* on page AppxE-6. No extension packets are defined for ARMv7-M, all packets reside in the primary A[4:0] packet ID address space.

For ARMv7-M, the following packet discriminator values are defined:

0	event counter wrapping
1	exception tracing
2	PC sampling
8 to 23	data tracing

E.2.1 Event Packets – Discriminator ID0

An event packet contains a set of bits to show which event counters have overflowed (wrapped). The packet is emitted when a counter overflows. Typically a single counter will overflow, however combinations of counter overflow can occur causing multiple bits to be set. The event counters each have an enable bit in the DWT_CTRL register. The event packet format is shown below:

Table E-10 Event packet (discriminator ID0) format

	bit [7]							bit [0]
byte 1	0	0	0	0	0	1	0	1
byte 2	x	x	Cyc	Fold	LSU	Sleep	Exc	CPI

The association of flags in the event packet and the DWT counters is summarized in Table E-11.

Table E-11 Event flag support

Flag	Controlled by	Counter	Interpretation
Cyc	CYCEVTENA (PCSAMPLENA == 0)	POSTCNT	Triggers when POSTCNT == 0, the cycle counter is enabled, and PC sampling is disabled.
Fold	FOLDEVTENA	FOLDCNT	Counts 0 cycle (folded) instructions
LSU	LSUEVTENA	LSUCNT	Counts LSU overhead.
Sleep	SLEEPEVTENA	SLEEPCNT	Counts cycles where the core is sleeping
Exc	EXCEVTENA	EXCCNT	Counts exception overhead
CPI	CPIEVTENA	CPICNT	Counts all non-LSU instruction overhead

For more information see *Cycle counter and PC sampling support* on page C1-34.

E.2.2 Exception Trace Packets – Discriminator ID1

An Interrupt trace packet is emitted whenever an exception is entered, exited (returned-from), and returned-to. The packet contains the exception number (see *Exception number definition* on page B1-16) and an indicator of which action it is (A). Exception tracing is enabled using the EXCTRCENA bit.

Table E-12 Event packet (discriminator ID1) format

	bit [7]							bit [0]
byte 1	0	0	0	0	1	1	1	0
byte 2	ExceptionNumber[7:0]							
byte 3	0	0	Fn<1>	Fn<0>	0	0	0	ExcNum[8]

Fn<1:0> is one of:

Value	Function (Fn)
00	Not used – invalid
01	Entry to ExceptionNumber
10	Exit from ExceptionNumber
11	Return to ExceptionNumber

E.2.3 PC Sample Packets – Discriminator ID2

PC samples are emitted at fixed time intervals (accurate to a clock cycle). The rate per second can be configured using the CYCTAP control field and the POSTCNT counter in DWT_CTRL (see *Control Register (DWT_CTRL)* on page C1-48). PC Sampling is enabled using PCSAMPLENA. Each sample contains the PC address or a sleep marker (if the processor is asleep).

The full PC packet is shown below.

Table E-13 Event packet (discriminator ID2) format

	bit [7]							bit [0]
byte 1	0	0	0	1	0	1	1	1
byte 2	PC[7:0] (halfword aligned)							
byte 3	PC[15:8]							
byte 4	PC[23:16]							
byte 5	PC[31:24]							

The sleep packet is shown below.

Table E-14 Sleep packet format

	bit [7]							bit [0]
byte 1	0	0	0	1	0	1	0	1
byte 2	0 - reserved							

E.2.4 Data Trace (Watchpoint) Packets – Discriminator ID8 to ID23

Data trace packets are emitted when a data trace directive as specified in the DWT_FUNCTIONx registers requires sampled data to be captured on a comparator match. For information on the comparator support, and the directives that generate data trace packets, see Table C1-20 on page C1-39 and Table C1-21 on page C1-41.

Note

The data trace packet protocol is currently only defined for DWT_FUNCTIONx registers, where 0 ≤ x ≤ 3.

The packet combinations that can be emitted are:

- PC-sample
- Data-value (what was read or written, including size).
- PC-sample followed by data-value.
- Address-offset followed by data-value.

The packet format for a data value is shown below.

Table E-15 Event packet (discriminator ID16 to ID23) format

	bit [7]							bit [0]
byte 1	1	0	COMP number[1:0]		WnR	1	S	S
byte 2	VALUE[7:0]							
byte 3	VALUE[15:8]							
byte 4	VALUE[23:16]							
byte 5	VALUE[31:24]							

Where the size may be 8, 16, or 32, depending on the access size (load or store size). WnR is 1 if write, 0 if read.

Packet format - data trace PC sampling

The packet format for a PC associated with a data trace is shown below.

Table E-16 Event packet (discriminator ID8, ID10, ID12, ID14) format

	bit [7]						bit [0]
byte 1	0	1	COMP number[1:0]	0	1	1	1
byte 2	PC[7:0]						
byte 3	PC[15:8]						
byte 4	PC[23:16]						
byte 5	PC[31:24]						

Packet format - data trace address offset

The packet format for a data address offset (when EMITRANGE=1) is shown below.

Table E-17 Event packet (discriminator ID9, ID11, ID13, ID15) format

	bit [7]						bit [0]
byte 1	0	1	COMP number[1:0]	1	1	1	0
byte 2	OFFSET[7:0]						
byte 3	OFFSET[15:8]						

Appendix F
ARMv7-R differences

While Thumb-2 technology is common across all the ARMv7 profiles, there are other key similarities between the ARMv7-M and ARMv7-R profiles. By understanding the similarities and differences, it is possible to minimize the effort in supporting software on both profiles, or to generate a system architecture allowing straightforward migration from one profile to the other.

A system tradeoff that needs to be made as part of the profile decision is absolute performance versus interrupt latency.

F.1 Endian support

ARMv7-R supports instruction fetches in big and little endian formats, using the IE bit in the ARMv7-R System Control register. ARMv7-M only supports instruction fetches in little endian format. Where a big endian instruction format is required with ARMv7-M, byte swapping within a halfword is required in the bus fabric. The byte swap is required for instruction fetches only and must not occur on data accesses.

By example, for instruction fetches over a 32-bit bus:

```
PrefetchInstr<31:24> -> PrefetchInstr<23:16>
PrefetchInstr<23:16> -> PrefetchInstr<31:24>
PrefetchInstr<15:8> -> PrefetchInstr<7:0>
PrefetchInstr<7:0> -> PrefetchInstr<15:8>
```

A configurable endian model (see *Endian support* on page A3-5) is supported by both ARMv7-M and ARMv7-R. While ARMv7-R supports dynamic endian control via a control bit in its xPSR and System Control register EE bit, ARMv7-M is statically configured on reset.

F.2 Application level support

At the application level, ARMv7-M can be considered as a subset of ARMv7-R. All the ARMv7-M application level instructions are supported in ARMv7-R, along with the same flag and general purpose register support. However, the LDM and STM instructions are always restartable in ARMv7-R and do not support the continuation (xPSR ICI bits) feature. Privileged access execution will expose the system level differences.

ARMv7-R has additional support for:

- SIMD instructions and saturated arithmetic. ARMv7-M only supports the SSAT and USAT saturation instructions.
- ARM and Thumb instruction set support. ARMv7-M is Thumb only.
- ARMv7-M only supports load and store exclusive instructions (bytes, halfwords and words) for synchronization. ARMv7-R supports LDREXD and STREXD too, along with the older swap (SWP and SWPB) instructions.

Both ARMv7-R and ARMv7-M support hardware divide (SDIV and UDIV) instructions, introduced to the architecture in these profiles.

F.3 System level support

The programmers' model is the key difference

- ARMv7-R supports banked registers and a modal system with fixed entry points (addresses) for exception handling. Control and configuration is through the System Coprocessor interface. Stacking and unstacking is under software control.
- ARMv7-M only banks the stack pointer. It uses a combination of special-purpose registers and memory mapped resources for system configuration and execution management. Auto-stacking/unstacking on exception entry and exit is a key difference from ARMv7-R.

System level instruction support is different, reflecting the different programmers' models. CPS, MRS and MSR are common instructions, but execute differently. There are additional system level instructions in ARMv7-R, for example SRS and RFE. WFE and WFI behavior differs too due to the exception model differences.

Both profiles support the Protected Memory System Architecture (PMSAv7), offering the same features sets where implemented. The register access mechanisms are different, however, the register layouts are generally the same with the notable exception of fault handling. Fault handling differences are due to the different exception models. When PMSAv7 is not supported or is disabled, both profiles have a default memory map. The default memory maps offer similar breakdown of memory with different attributes, but the maps are not identical.

ARMv7-R is designed for higher performance (higher clock rate) parts and includes support for closely coupled caches. ARMv7-M only supports memory-mapped system caches.

Interrupt control is an integral part of the ARMv7-M exception model. While not part of the ARMv7-R architecture, ARM's General Interrupt Controller (GIC) offers a common prioritization and interrupt handling model to ARMv7-M. Use of a GIC with an ARMv7-R processor mitigates many of the exception model differences.

ARMv7-M defines a system timer. A similar timer can be used with ARMv7-R, and its interrupt routed through a GIC for maximum compatibility.

F.4 Debug support

Both profiles logically support halting and monitor debug. The mechanisms for breakpoint and watchpoint handling are different. There are also different degrees of counter support for profiling. Both support an optional trace (ETM) feature. ARMv7-M is generally less invasive in its debug support, and offers additional software and hardware event generation trace capabilities as part of the basic architecture.

Appendix G
Pseudocode definition

This appendix provides a formal definition of the pseudocode used in this book, and lists the *helper* procedures and functions used by pseudocode to perform useful architecture-specific jobs. It contains the following sections:

G.1 Instruction encoding diagrams and pseudocode

Instruction descriptions in this book contain:

- An Encoding section, containing one or more encoding diagrams, each followed by some encoding-specific pseudocode that translates the fields of the encoding into inputs for the common pseudocode of the instruction, and picks out any encoding-specific special cases.
- An Operation section, containing common pseudocode that applies to all of the encodings being described. The Operation section pseudocode contains a call to the `EncodingSpecificOperations()` function, either at its start or after only a condition check performed by `if ConditionPassed() then`.

An encoding diagram specifies each bit of the instruction as one of the following:

- An obligatory 0 or 1, represented in the diagram as 0 or 1. If this bit does not have this value, the encoding corresponds to a different instruction.
- A *should be* 0 or 1, represented in the diagram as (0) or (1). If this bit does not have this value, the instruction is UNPREDICTABLE.
- A named single bit or a bit within a named multi-bit field.

An encoding diagram matches an instruction if all obligatory bits are identical in the encoding diagram and the instruction.

The execution model for an instruction is:

1. Find all encoding diagrams that match the instruction. It is possible that no encoding diagrams match. In that case, abandon this execution model and consult the relevant instruction set chapter instead to find out how the instruction is to be treated. (The bit pattern of such an instruction is usually reserved and UNDEFINED, though there are some other possibilities. For example, unallocated hint instructions are documented as being reserved and to be executed as NOPs.)
2. If the operation pseudocode for the matching encoding diagrams starts with a condition check, perform that condition check. If the condition check fails, abandon this execution model and treat the instruction as a NOP. (If there are multiple matching encoding diagrams, either all or none of their corresponding pieces of common pseudocode start with a condition check.)
3. Perform the encoding-specific pseudocode for each of the matching encoding diagrams independently and in parallel. Each such piece of encoding-specific pseudocode starts with a bitstring variable for each named bit or multi-bit field within its corresponding encoding diagram, named the same as the bit or multi-bit field and initialized with the values of the corresponding bit(s) from the bit pattern of the instruction.

   In a few cases, the encoding diagram contains more than one bit or field with the same name. When this occurs, the values of all of those bits or fields are expected to be identical, and the encoding-specific pseudocode contains a special case using the `Consistent()` function to specify what happens if this is not the case. This function returns `TRUE` if all instruction bits or fields with the same name as its argument have the same value, and `FALSE` otherwise.

If there are multiple matching encoding diagrams, all but one of the corresponding pieces of pseudocode must contain a special case that indicates that it does not apply. Discard the results of all such pieces of pseudocode and their corresponding encoding diagrams.

There is now one remaining piece of pseudocode and its corresponding encoding diagram left to consider. This pseudocode might also contain a special case (most commonly one indicating that it is UNPREDICTABLE). If so, abandon this execution model and treat the instruction according to the special case.

4. Check the *should be* bits of the encoding diagram against the corresponding bits of the bit pattern of the instruction. If any of them do not match, abandon this execution model and treat the instruction as UNPREDICTABLE.

5. Perform the rest of the operation pseudocode for the instruction description that contains the encoding diagram. That pseudocode starts with all variables set to the values they were left with by the encoding-specific pseudocode.

The `ConditionPassed()` call in the common pseudocode (if present) performs step 2, and the `EncodingSpecificOperations()` call performs steps 3 and 4.

G.1.1 Pseudocode

The pseudocode provides precise descriptions of what instructions do. Instruction fields are referred to by the names shown in the encoding diagram for the instruction.

The pseudocode is described in detail in the following sections.

G.2 Limitations of pseudocode

The pseudocode descriptions of instruction functionality have a number of limitations. These are mainly due to the fact that, for clarity and brevity, the pseudocode is a sequential and mostly deterministic language.

These limitations include:

- Pseudocode does not describe the ordering requirements when an instruction generates multiple memory accesses. For a description of the ordering requirements on memory accesses see *Memory access order* on page A3-30.
- Pseudocode does not describe the exact rules when an UNDEFINED instruction fails its condition check. In such cases, the `UNDEFINED` pseudocode statement lies inside the `if ConditionPassed() then ...` structure, either directly or within the `EncodingSpecificOperations()` function call, and so the pseudocode indicates that the instruction executes as a NOP. See *Conditional execution of undefined instructions* on page A6-9 for more information.
- The pseudocode statements `UNDEFINED`, `UNPREDICTABLE` and `SEE` indicate behavior that differs from that indicated by the pseudocode being executed. If one of them is encountered:
  - — Earlier behavior indicated by the pseudocode is only specified as occurring to the extent required to determine that the statement is executed.
  - — No subsequent behavior indicated by the pseudocode occurs. This means that these statements terminate pseudocode execution.

  For more information see *Simple statements* on page AppxG-17.

G.3 Data Types

This section describes:

- *General data type rules*
- *Bitstrings*
- *Integers* on page AppxG-6
- *Reals* on page AppxG-6
- *Booleans* on page AppxG-6
- *Enumerations* on page AppxG-6
- *Lists* on page AppxG-7
- *Arrays* on page AppxG-8.

G.3.1 General data type rules

ARM Architecture pseudocode is a strongly-typed language. Every constant and variable is of one of the following types:

- bitstring
- integer
- boolean
- real
- enumeration
- list
- array.

The type of a constant is determined by its syntax. The type of a variable is normally determined by assignment to the variable, with the variable being implicitly declared to be of the same type as whatever is assigned to it. For example, the assignments `x = 1`, `y = '1'`, and `z = TRUE` implicitly declare the variables `x`, `y` and `z` to have types integer, length-1 bitstring and boolean respectively.

Variables can also have their types declared explicitly by preceding the variable name with the name of the type. This is most often done in function definitions for the arguments and the result of the function.

These data types are described in more detail in the following sections.

G.3.2 Bitstrings

A bitstring is a finite-length string of 0s and 1s. Each length of bitstring is a different type. The minimum allowed length of a bitstring is 1.

The type name for bitstrings of length `N` is `bits(N)`. A synonym of `bits(1)` is `bit`.

Bitstring constants are written as a single quotation mark, followed by the string of 0s and 1s, followed by another single quotation mark. For example, the two constants of type `bit` are `'0'` and `'1'`. Spaces can be included in the bitstring for clarity.

A special form of bitstring constant with 'x' bits is permitted in bitstring comparisons. See *Equality and non-equality testing* on page AppxG-11 for details.

Every bitstring value has a left-to-right order, with the bits being numbered in standard *little-endian* order. That is, the leftmost bit of a bitstring of length N is bit N-1 and its rightmost bit is bit 0. This order is used as the most-significant-to-least-significant bit order in conversions to and from integers. For bitstring constants and bitstrings derived from encoding diagrams, this order matches the way they are printed.

Bitstrings are the only concrete data type in pseudocode, in the sense that they correspond directly to the contents of registers, memory locations, instructions, and so on. All of the remaining data types are abstract.

G.3.3 Integers

Pseudocode integers are unbounded in size and can be either positive or negative. That is, they are mathematical integers rather than what computer languages and architectures commonly call integers. Computer integers are represented in pseudocode as bitstrings of the appropriate length, associated with suitable functions to interpret those bitstrings as integers.

The type name for integers is `integer`.

Integer constants are normally written in decimal, such as `0`, `15`, `-1234`. They can also be written in C-style hexadecimal, such as `0x55` or `0x80000000`. Hexadecimal integer constants are treated as positive unless they have a preceding minus sign. For example, `0x80000000` is the integer $+2^{31}$. If -2^{31} needs to be written in hexadecimal, it should be written as `-0x80000000`.

G.3.4 Reals

Pseudocode reals are unbounded in size and precision. That is, they are mathematical real numbers, not computer floating-point numbers. Computer floating-point numbers are represented in pseudocode as bitstrings of the appropriate length, associated with suitable functions to interpret those bitstrings as reals.

The type name for reals is `real`.

Real constants are written in decimal with a decimal point (so `0` is an integer constant, but `0.0` is a real constant).

G.3.5 Booleans

A boolean is a logical true or false value.

The type name for booleans is `boolean`. This is not the same type as `bit`, which is a length-1 bitstring.

Boolean constants are `TRUE` and `FALSE`.

G.3.6 Enumerations

An enumeration is a defined set of symbolic constants, such as:

```
enumeration SRType (SRType_None, SRType_LSL, SRType_LSR,
                    SRType_ASR, SRType_ROR, SRType_RRX);
```

An enumeration always contains at least one symbolic constant, and symbolic constants are not allowed to be shared between enumerations.

Enumerations must be declared explicitly, though a variable of an enumeration type can be declared implicitly as usual by assigning one of the symbolic constants to it. By convention, each of the symbolic constants starts with the name of the enumeration followed by an underscore. The name of the enumeration is its type name, and the symbolic constants are its possible constants.

Note

Booleans are basically a pre-declared enumeration:

```
enumeration boolean {FALSE, TRUE};
```

that does not follow the normal naming convention and that has a special role in some pseudocode constructs, such as `if` statements.

G.3.7 Lists

A list is an ordered set of other data items, separated by commas and enclosed in parentheses, such as:

```
(bits(32) shifter_result, bit shifter_carry_out)
```

A list always contains at least one data item.

Lists are often used as the return type for a function that returns multiple results. For example, this particular list is the return type of the function `Shift_C()` that performs a standard ARM shift or rotation, when its first operand is of type `bits(32)`.

Some specific pseudocode operators use lists surrounded by other forms of bracketing than parentheses. These are:

- Bitstring extraction operators, which use lists of bit numbers or ranges of bit numbers surrounded by angle brackets "<...>".
- Array indexing, which uses lists of array indexes surrounded by square brackets "[...]".
- Array-like function argument passing, which uses lists of function arguments surrounded by square brackets "[...]".

Each combination of data types in a list is a separate type, with type name given by just listing the data types (that is, `(bits(32),bit)` in the above example). The general principle that types can be declared by assignment extends to the types of the individual list items within a list. For example:

```
(shift_t, shift_n) = ('00', 0);
```

implicitly declares `shift_t`, `shift_n` and `(shift_t,shift_n)` to be of types `bits(2)`, `integer` and `(bits(2),integer)` respectively.

A list type can also be explicitly named, with explicitly named elements in the list. For example:

```
type ShiftSpec is (bits(2) shift, integer amount);
```

After this definition and the declaration:

```
ShiftSpec abc;
```

the elements of the resulting list can then be referred to as "abc.shift" and "abc.amount". This sort of qualified naming of list elements is only permitted for variables that have been explicitly declared, not for those that have been declared by assignment only.

Explicitly naming a type does not alter what type it is. For example, after the above definition of ShiftSpec, ShiftSpec and (bits(2),integer) are two different names for the same type, not the names of two different types. In order to avoid ambiguity in references to list elements, it is an error to declare a list variable multiple times using different names of its type or to qualify it with list element names not associated with the name by which it was declared.

An item in a list that is being assigned to may be written as "-" to indicate that the corresponding item of the assigned list value is discarded. For example:

```
(shifted, -) = LSL_C(operand, amount);
```

List constants are written as a list of constants of the appropriate types, like ('00', 0) in the above example.

G.3.8 Arrays

Pseudocode arrays are indexed by either enumerations or integer ranges (represented by the lower inclusive end of the range, then "..", then the upper inclusive end of the range). For example:

```
enumeration PhysReg {
    PhysReg_R0,    PhysReg_R1,     PhysReg_R2,    PhysReg_R3,
    PhysReg_R4,    PhysReg_R5,     PhysReg_R6,    PhysReg_R7,
    PhysReg_R8,    PhysReg_R9,     PhysReg_R10,   PhysReg_R11,
    PhysReg_R12,   PhysReg_SP_Process,            PhysReg_SP_Main,
    PhysReg_LR,    PhysReg_PC};

array bits(32) _R[PhysReg];

array bits(8) _Memory[0..0xFFFFFFFF];
```

Arrays are always explicitly declared, and there is no notation for a constant array. Arrays always contain at least one element, because enumerations always contain at least one symbolic constant and integer ranges always contain at least one integer.

Arrays do not usually appear directly in pseudocode. The items that syntactically look like arrays in pseudocode are usually array-like functions such as R[i], MemU[address,size] or Element[i,type]. These functions package up and abstract additional operations normally performed on accesses to the underlying arrays, such as register banking, memory protection, endian-dependent byte ordering, exclusive-access housekeeping and vector element processing.

G.4 Expressions

This section describes:

- *General expression syntax*
- *Operators and functions - polymorphism and prototypes* on page AppxG-10
- *Precedence rules* on page AppxG-10.

G.4.1 General expression syntax

An expression is one of the following:

- a constant
- a variable, optionally preceded by a data type name to declare its type
- the word `UNKNOWN` preceded by a data type name to declare its type
- the result of applying a language-defined operator to other expressions
- the result of applying a function to other expressions.

Variable names normally consist of alphanumeric and underscore characters, starting with an alphabetic or underscore character.

Each register described in the text is to be regarded as declaring a correspondingly named bitstring variable, and that variable has the stated behavior of the register. For example, if a bit of a register is stated to read as 0 and ignore writes, then the corresponding bit of its variable reads as 0 and ignore writes.

An expression like "`bits(32) UNKNOWN`" indicates that the result of the expression is a value of the given type, but the architecture does not specify what value it is and software must not rely on such values. The value produced must not constitute a security hole and must not be promoted as providing any useful information to software. (This was called an UNPREDICTABLE value in previous ARM Architecture documentation. It is related to but not the same as `UNPREDICTABLE`, which says that the entire architectural state becomes similarly unspecified.)

A subset of expressions are assignable. That is, they can be placed on the left-hand side of an assignment. This subset consists of:

- Variables
- The results of applying some operators to other expressions. The description of each language-defined operator that can generate an assignable expression specifies the circumstances under which it does so. (For example, those circumstances might include one or more of the expressions the operator operates on themselves being assignable expressions.)
- The results of applying array-like functions to other expressions. The description of an array-like function specifies the circumstances under which it can generate an assignable expression.

Every expression has a data type. This is determined by:

- For a constant, the syntax of the constant.
- For a variable, there are three possible sources for the type
  - its optional preceding data type name

- — a data type it was given earlier in the pseudocode by recursive application of this rule
- — a data type it is being given by assignment (either by direct assignment to it, or by assignment to a list of which it is a member).

It is a pseudocode error if none of these data type sources exists for a variable, or if more than one of them exists and they do not agree about the type.

- For a language-defined operator, the definition of the operator.
- For a function, the definition of the function.

G.4.2 Operators and functions - polymorphism and prototypes

Operators and functions in pseudocode can be polymorphic, producing different functionality when applied to different data types. Each of the resulting forms of an operator or function has a different prototype definition. For example, the operator `+` has forms that act on various combinations of integers, reals and bitstrings.

One particularly common form of polymorphism is between bitstrings of different lengths. This is represented by using `bits(N)`, `bits(M)`, and so on, in the prototype definition.

G.4.3 Precedence rules

The precedence rules for expressions are:

1. Constants, variables and function invocations are evaluated with higher priority than any operators using their results.
2. Expressions on integers follow the normal *exponentiation before multiply/divide before add/subtract* operator precedence rules, with sequences of multiply/divides or add/subtracts evaluated left-to-right.
3. Other expressions must be parenthesized to indicate operator precedence if ambiguity is possible, but need not be if all allowable precedence orders under the type rules necessarily lead to the same result. For example, if `i`, `j` and `k` are integer variables, `i > 0 && j > 0 && k > 0` is acceptable, but `i > 0 && j > 0 || k > 0` is not.

G.5 Operators and built-in functions

This section describes:

- *Operations on generic types*
- *Operations on booleans*
- *Bitstring manipulation*
- *Arithmetic* on page AppxG-14.

G.5.1 Operations on generic types

The following operations are defined for all types.

Equality and non-equality testing

Any two values `x` and `y` of the same type can be tested for equality by the expression `x == y` and for non-equality by the expression `x != y`. In both cases, the result is of type `boolean`.

A special form of comparison with a bitstring constant that includes `'x'` bits as well as `'0'` and `'1'` bits is permitted. The bits corresponding to the `'x'` bits are ignored in determining the result of the comparison. For example, if opcode is a 4-bit bitstring, `opcode == '1x0x'` is equivalent to `opcode<3> == '1' && opcode<1> == '0'`. This special form is also permitted in the implied equality comparisons in `when` parts of `case ... of ...` structures.

Conditional selection

If `x` and `y` are two values of the same type and `t` is a value of type `boolean`, then `if t then x else y` is an expression of the same type as `x` and `y` that produces `x` if `t` is `TRUE` and `y` if `t` is `FALSE`.

G.5.2 Operations on booleans

If `x` is a `boolean`, then `!x` is its logical inverse.

If `x` and `y` are booleans, then `x && y` is the result of ANDing them together. As in the C language, if `x` is `FALSE`, the result is determined to be `FALSE` without evaluating `y`.

If `x` and `y` are booleans, then `x || y` is the result of ORing them together. As in the C language, if `x` is `TRUE`, the result is determined to be `TRUE` without evaluating `y`.

If `x` and `y` are booleans, then `x ^ y` is the result of exclusive-ORing them together.

G.5.3 Bitstring manipulation

The following bitstring manipulation functions are defined:

Bitstring length and top bit

If x is a bitstring, the bitstring length function Len(x) returns its length as an integer, and TopBit(x) is the leftmost bit of x (= x<Len(x)-1> using bitstring extraction.

Bitstring concatenation and replication

If x and y are bitstrings of lengths N and M respectively, then x:y is the bitstring of length N+M constructed by concatenating x and y in left-to-right order.

If x is a bitstring and n is an integer with n > 0, Replicate(x,n) is the bitstring of length n*Len(x) consisting of n copies of x concatenated together and:

- Zeros(n) = Replicate('0',n)
- Ones(n) = Replicate('1',n)

Bitstring extraction

The bitstring extraction operator extracts a bitstring from either another bitstring or an integer. Its syntax is x<integer_list>, where x is the integer or bitstring being extracted from, and <integer_list> is a list of integers enclosed in angle brackets rather than the usual parentheses. The length of the resulting bitstring is equal to the number of integers in <integer_list>.

In x<integer_list>, each of the integers in <integer_list> must be:

- >= 0
- < Len(x) if x is a bitstring.

The definition of x<integer_list> depends on whether integer_list contains more than one integer. If it does, x<i,j,k,...,n> is defined to be the concatenation:

```
x<i> : x<j> : x<k> : ... : x<n>
```

If integer_list consists of just one integer i, x<i> is defined to be:

- if x is a bitstring, '0' if bit i of x is a zero and '1' if bit i of x is a one.
- if x is an integer, let y be the unique integer in the range 0 to 2^(i+1)-1 that is congruent to x modulo 2^(i+1). Then **x<i>** is '0' if y < 2^i and '1' if y >= 2^i.

Loosely, this second definition treats an integer as equivalent to a sufficiently long 2's complement representation of it as a bitstring.

In <integer_list>, the notation i:j with i >= j is shorthand for the integers in order from i down to j, both ends inclusive. For example, instr<31:28> is shorthand for instr<31,30,29,28>.

The expression x<integer_list> is assignable provided x is an assignable bitstring and no integer appears more than once in <integer_list>. In particular, x<i> is assignable if x is an assignable bitstring and 0 <= i < Len(x).

Encoding diagrams for registers frequently show named bits or multi-bit fields. For example, the encoding diagram for the APSR shows its bit<31> as N. In such cases, the syntax APSR.N is used as a more readable synonym for APSR<31>.

Logical operations on bitstrings

If `x` is a bitstring, `NOT(x)` is the bitstring of the same length obtained by logically inverting every bit of `x`.

If `x` and `y` are bitstrings of the same length, `x AND y`, `x OR y`, and `x EOR y` are the bitstrings of that same length obtained by logically ANDing, ORing, and exclusive-ORing corresponding bits of `x` and `y` together.

Bitstring count

If x is a bitstring, `BitCount(x)` produces an integer result equal to the number of bits of `x` that are ones.

Testing a bitstring for being all zero or all ones

If `x` is a bitstring, `IsZero(x)` produces `TRUE` if all of the bits of `x` are zeros and `FALSE` if any of them are ones, and `IsZeroBit(x)` produces `'1'` if all of the bits of `x` are zeros and `'0'` if any of them are ones. IsOnes(x) and IsOnesBit(x) work in the corresponding way. So:

```
IsZero(x)    = (BitCount(x) == 0)

IsOnes(x)    = (BitCount(x) == Len(x))

IsZeroBit(x) = if IsZero(x) then '1' else '0'

IsOnesBit(x) = if IsOnes(x) then '1' else '0'
```

Lowest and highest set bits of a bitstring

If `x` is a bitstring, and N = Len(x):

- `LowestSetBit(x)` is the minimum bit number of any of its bits that are ones. If all of its bits are zeros, `LowestSetBit(x) = N`.
- `HighestSetBit(x)` is the maximum bit number of any of its bits that are ones. If all of its bits are zeros, `HighestSetBit(x) = -1`.
- `CountLeadingZeroBits(x) = N - 1 - HighestSetBit(x)` is the number of zero bits at the left end of x, in the range 0 to N.
- `CountLeadingSignBits(x) = CountLeadingZeroBits(x<N-1:1> EOR x<N-2:0>)` is the number of copies of the sign bit of x at the left end of x, excluding the sign bit itself, and is in the range 0 to N-1.

Zero-extension and sign-extension of bitstrings

If `x` is a bitstring and `i` is an integer, then `ZeroExtend(x,i)` is `x` extended to a length of `i` bits, by adding sufficient zero bits to its left. That is, if `i == Len(x)`, then `ZeroExtend(x,i) = x`, and if `i > Len(x)`, then:

```
ZeroExtend(x,i) = Zeros(i-Len(x)) : x
```

If x is a bitstring and i is an integer, then SignExtend(x,i) is x extended to a length of i bits, by adding sufficient copies of its leftmost bit to its left. That is, if i == Len(x), then SignExtend(x,i) = x, and if i > Len(x), then:

```
SignExtend(x,i) = Replicate(TopBit(x), i-Len(x)) : x
```

It is a pseudocode error to use either ZeroExtend(x,i) or SignExtend(x,i) in a context where it is possible that i < Len(x).

Converting bitstrings to integers

If x is a bitstring, SInt(x) is the integer whose 2's complement representation is x:

```
// SInt()
// ======

integer SInt(bits(N) x)
    result = 0;
    for i = 0 to N-1
        if x<i> == ’1’ then result = result + 2^i;
    if x<N-1> == ’1’ then result = result - 2^N;
    return result;
```

UInt(x) is the integer whose unsigned representation is x:

```
// UInt()
// ======

integer UInt(bits(N) x)
    result = 0;
    for i = 0 to N-1
        if x<i> == ’1’ then result = result + 2^i;
    return result;
```

Int(x, unsigned) returns either SInt(x) or UInt(x) depending on the value of its second argument:

```
// Int()
// =====

integer Int(bits(N) x, boolean unsigned)
    result = if unsigned then UInt(x) else SInt(x);
    return result;
```

G.5.4 Arithmetic

Most pseudocode arithmetic is performed on integer or real values, with operands being obtained by conversions from bitstrings and results converted back to bitstrings afterwards. As these data types are the unbounded mathematical types, no issues arise about overflow or similar errors.

Unary plus, minus and absolute value

If x is an integer or real, then +x is x unchanged, -x is x with its sign reversed, and ABS(x) is the absolute value of x. All three are of the same type as x.

Addition and subtraction

If x and y are integers or reals, x+y and x-y are their sum and difference. Both are of type integer if x and y are both of type integer, and real otherwise.

Addition and subtraction are particularly common arithmetic operations in pseudocode, and so it is also convenient to have definitions of addition and subtraction acting directly on bitstring operands.

If x and y are bitstrings of the same length N = Len(x) = Len(y), then x+y and x-y are the least significant N bits of the results of converting them to integers and adding or subtracting them. Signed and unsigned conversions produce the same result:

```
x+y = (SInt(x) + SInt(y))<N-1:0>
    = (UInt(x) + UInt(y))<N-1:0>

x-y = (SInt(x) - SInt(y))<N-1:0>
    = (UInt(x) - UInt(y))<N-1:0>
```

If x is a bitstring of length N and y is an integer, x+y and x-y are the bitstrings of length N defined by x+y = x + y<N-1:0> and x-y = x - y<N-1:0>. Similarly, if x is an integer and y is a bitstring of length M, x+y and x-y are the bitstrings of length M defined by x+y = x<M-1:0> + y and x-y = x<M-1:0> - y.

Comparisons

If x and y are integers or reals, then x == y, x != y, x < y, x <= y, x > y, and x >= y are equal, not equal, less than, less than or equal, greater than, and greater than or equal comparisons between them, producing boolean results. In the case of == and !=, this extends the generic definition applying to any two values of the same type to also act between integers and reals.

Multiplication

If x and y are integers or reals, then x * y is the product of x and y, of type integer if both x and y are of type integer and otherwise of type real.

Division and modulo

If x and y are integers or reals, then x / y is the result of dividing x by y, and is always of type real.

If x and y are integers, then x DIV y and x MOD y are defined by:

```
x DIV y = RoundDown(x / y)
x MOD y = x - y * (x DIV y)
```

It is a pseudocode error to use any x / y, x MOD y, or x DIV y in any context where y can be zero.

Square Root

If `x` is an integer or a real, `Sqrt(x)` is its square root, and is always of type real.

Rounding and aligning

If x is a real:

- `RoundDown(x)` produces the largest integer `n` such that `n <= x`.
- `RoundUp(x)` produces the smallest integer `n` such that `n >= x`.
- `RoundTowardsZero(x)` produces `RoundDown(x)` if `x > 0.0`, `0 if x == 0.0`, and `RoundUp(x) if x < 0.0`.

If `x` and `y` are integers, `Align(x,y) = y * (x DIV y)` is an integer.

If `x` is a bitstring and `y` is an integer, `Align(x,y) = (Align(UInt(x),y))<Len(x)-1:0>` is a bitstring of the same length as `x`.

It is a pseudocode error to use either form of `Align(x,y)` in any context where `y` can be 0. In practice, `Align(x,y)` is only used with `y` a constant power of two, and the bitstring form used with `y = 2^n` has the effect of producing its argument with its n low-order bits forced to zero.

Scaling

If `n` is an integer, `2^n` is the result of raising `2` to the power `n` and is of type `real`.

If `x` and `n` are integers, then:

- `x << n = RoundDown(x * 2^n)`
- `x >> n = RoundDown(x * 2^(-n))`.

Maximum and minimum

If `x` and `y` are integers or reals, then `Max(x,y)` and `Min(x,y)` are their maximum and minimum respectively. Both are of type `integer` if both `x` and `y` are of type `integer` and of type `real` otherwise.

G.6 Statements and program structure

This section describes the control statements used in the pseudocode.

G.6.1 Simple statements

The following simple statements must all be terminated with a semicolon, as shown.

Assignments

An assignment statement takes the form:

```
<assignable_expression> = <expression>;
```

Procedure calls

A procedure call takes the form:

```
<procedure_name>(<arguments>);
```

Return statements

A procedure return takes the form:

```
return;
```

and a function return takes the form:

```
return <expression>;
```

where `<expression>` is of the type the function prototype line declared.

UNDEFINED

The statement:

```
UNDEFINED;
```

indicates a special case that replaces the behavior defined by the current pseudocode (apart from behavior required to determine that the special case applies). The replacement behavior is that the Undefined Instruction exception is taken.

UNPREDICTABLE

The statement:

```
UNPREDICTABLE;
```

indicates a special case that replaces the behavior defined by the current pseudocode (apart from behavior required to determine that the special case applies). The replacement behavior is not architecturally defined and must not be relied upon by software. It must not constitute a security hole or halt or hang the system, and must not be promoted as providing any useful information to software.

SEE...

The statement:

```
SEE <reference>;
```

indicates a special case that replaces the behavior defined by the current pseudocode (apart from behavior required to determine that the special case applies). The replacement behavior is that nothing occurs as a result of the current pseudocode because some other piece of pseudocode defines the required behavior. The `<reference>` indicates where that other pseudocode can be found.

IMPLEMENTATION_DEFINED

The statement:

```
IMPLEMENTATION_DEFINED <text>;
```

indicates a special case that specifies that the behavior is IMPLEMENTATION DEFINED. Following text can give more information.

SUBARCHITECTURE_DEFINED

The statement:

```
SUBARCHITECTURE_DEFINED <text>;
```

indicates a special case that specifies that the behavior is SUBARCHITECTURE DEFINED. Following text can give more information.

G.6.2 Compound statements

Indentation is normally used to indicate structure in compound statements. The statements contained in structures such as `if ... then ... else ...` or procedure and function definitions are indented more deeply than the statement itself, and their end is indicated by returning to the original indentation level or less.

Indentation is normally done by four spaces for each level.

if ... then ... else ...

A multi-line `if ... then ... else ...` structure takes the form:

```
if <boolean_expression> then
    <statement 1>
    <statement 2>
    ...
    <statement n>
elsif <boolean_expression> then
    <statement a>
    <statement b>
    ...
    <statement z>
else
    <statement A>
    <statement B>
    ...
    <statement Z>
```

The `else` and its following statements are optional.

```
if <boolean_expression> then
    <statement 1>
    <statement 2>
    ...
    <statement n>
elsif <boolean_expression> then
    <statement a>
    <statement b>
    ...
    <statement z>
else
    <statement A>
    <statement B>
    ...
    <statement Z>
```

The block of lines consisting of `elsif` and its indented statements is optional, and multiple such blocks can be used.

The block of lines consisting of `else` and its indented statements is optional.

Abbreviated one-line forms can be used when there are only simple statements in the `then` part and (if present) the `else` part, as follows:

```
if <boolean_expression> then <statement 1>

if <boolean_expression> then <statement 1> else <statement A>

if <boolean_expression> then <statement 1> <statement 2> else <statement A>
```

Note

In these forms, `<statement 1>`, `<statement 2>` and `<statement A>` must be terminated by semicolons. This and the fact that the `else` part is optional are differences from the `if ... then ... else ...` expression.

repeat ... until ...

A `repeat ... until ...` structure takes the form:

```
repeat
    <statement 1>
    <statement 2>
    ...
    <statement n>
until <boolean_expression>;
```

while ... do

A `while ... do` structure takes the form:

```
while <boolean_expression> do
    <statement 1>
    <statement 2>
    ...
    <statement n>
```

for ...

A `for ...` structure takes the form:

```
for <assignable_expression> = <integer_expr1> to <integer_expr2>
    <statement 1>
    <statement 2>
    ...
    <statement n>
```

case ... of ...

A `case ... of ...` structure takes the form:

```
case <expression> of
    when <constant values>
        <statement 1>
        <statement 2>
        ...
        <statement n>
    ... more "when" groups ...
    otherwise
        <statement A>
        <statement B>
        ...
        <statement Z>
```

where `<constant values>` consists of one or more constant values of the same type as `<expression>`, separated by commas. Abbreviated one line forms of `when` and `otherwise` parts can be used when they contain only simple statements.

If `<expression>` has a bitstring type, `<constant values>` can also include bitstring constants containing `'x'` bits. See *Equality and non-equality testing* on page AppxG-11 for details.

Procedure and function definitions

A procedure definition takes the form:

```
<procedure name>(<argument prototypes>)
    <statement 1>
    <statement 2>
    ...
    <statement n>
```

where the `<argument prototypes>` consists of zero or more argument definitions, separated by commas. Each argument definition consists of a type name followed by the name of the argument.

Note

This first prototype line is not terminated by a semicolon. This helps to distinguish it from a procedure call.

A function definition is similar, but also declares the return type of the function:

```
<return type> <function name>(<argument prototypes>)
    <statement 1>
    <statement 2>
    ...
    <statement n>
```

An array-like function is similar, but with square brackets:

```
<return type> <function name>[<argument prototypes>]
    <statement 1>
    <statement 2>
    ...
    <statement n>
```

An array-like function also usually has an assignment prototype:

```
<function name>[<argument prototypes>] = <value prototypes>
    <statement 1>
    <statement 2>
    ...
    <statement n>
```

G.6.3 Comments

Two styles of pseudocode comment exist:

- `//` starts a comment that is terminated by the end of the line.
- `/*` starts a comment that is terminated by `*/`.

G.7 Miscellaneous helper procedures and functions

The functions described in this section are not part of the pseudocode specification. They are *helper* procedures and functions used by pseudocode to perform useful architecture-specific jobs. Each has a brief description and a pseudocode prototype. Some have had a pseudocode definition added.

G.7.1 ArchVersion()

This function returns the major version number of the architecture.

```
integer ArchVersion()
```

G.7.2 BadReg()

This function performs the check for the register numbers 13 and 15 that are disallowed for many Thumb register specifiers.

```
boolean BadReg(integer n)
return n == 13 || n == 15;
```

G.7.3 BreakPoint()

This procedure causes a debug breakpoint to occur.

G.7.4 CallSupervisor()

In the M profile, this procedure causes an SVCall exception.

G.7.5 ConditionPassed()

This function performs the condition test for an instruction, based on:

- the two Thumb conditional branch encodings (encodings T1 andT3 of the B instruction)
- the current values of the xPSR.IT[7:0] bits for other Thumb instructions.

```
boolean ConditionPassed()
```

G.7.6 Coproc_Accepted()

This function determines whether a coprocessor accepts an instruction.

```
boolean Coproc_Accepted(integer cp_num, bits(32) instr)
```

G.7.7 Coproc_DoneLoading()

This function determines for an `LDC` instruction whether enough words have been loaded.

```
boolean Coproc_DoneLoading(integer cp_num, bits(32) instr)
```

G.7.8 Coproc_DoneStoring()

This function determines for an STC instruction whether enough words have been stored.

```
boolean Coproc_DoneStoring(integer cp_num, bits(32) instr)
```

G.7.9 Coproc_GetOneWord()

This function obtains the word for an MRC instruction from the coprocessor.

```
bits(32) Coproc_GetOneWord(integer cp_num, bits(32) instr)
```

G.7.10 Coproc_GetTwoWords()

This function obtains the two words for an MRRC instruction from the coprocessor.

```
(bits(32), bits(32)) Coproc_GetTwoWords(integer cp_num, bits(32) instr)
```

G.7.11 Coproc_GetWordToStore()

This function obtains the next word to store for an STC instruction from the coprocessor

```
bits(32) Coproc_GetWordToStore(integer cp_num, bits(32) instr)
```

G.7.12 Coproc_InternalOperation()

This procedure instructs a coprocessor to perform the internal operation requested by a CDP instruction.

```
Coproc_InternalOperation(integer cp_num, bits(32) instr)
```

G.7.13 Coproc_SendLoadedWord()

This procedure sends a loaded word for an LDC instruction to the coprocessor.

```
Coproc_SendLoadedWord(bits(32) word, integer cp_num, bits(32) instr)
```

G.7.14 Coproc_SendOneWord()

This procedure sends the word for an MCR instruction to the coprocessor.

```
Coproc_SendOneWord(bits(32) word, integer cp_num, bits(32) instr)
```

G.7.15 Coproc_SendTwoWords()

This procedure sends the two words for an MCRR instruction to the coprocessor.

```
Coproc_SendTwoWords(bits(32) word1, bits(32) word2, integer cp_num,
                    bits(32) instr)
```

G.7.16 DataMemoryBarrier()

This procedure produces a Data Memory Barrier.

```
DataMemoryBarrier(bits(4) option)
```

G.7.17 DataSynchronizationBarrier()

This procedure produces a Data Synchronization Barrier.

```
DataSynchronizationBarrier(bits(4) option)
```

G.7.18 EncodingSpecificOperations()

This procedure invokes the encoding-specific pseudocode for an instruction encoding and checks the ’should be’ bits of the encoding, as described in *Instruction encoding diagrams and pseudocode* on page AppxG-2.

G.7.19 GenerateCoprocessorException()

This procedure raises a UsageFault exception for a rejected coprocessor instruction.

G.7.20 GenerateIntegerZeroDivide()

This procedure raises the appropriate exception for a division by zero in the integer division instructions `SDIV` and `UDIV`.

In the M profile, this is a UsageFault exception.

G.7.21 Hint_Debug()

This procedure supplies a hint to the debug system.

```
Hint_Debug(bits(4) option)
```

G.7.22 Hint_PreloadData()

This procedure performs a *preload data* hint.

```
Hint_PreloadData(bits(32) address)
```

~~G.7.23 Hint_PreloadDataForWrite()~~

~~This procedure performs a *preload data* hint with a probability that the use will be for a write.~~

```
Hint_PreloadDataForWrite(bits(32) address)
```

G.7.24 Hint_PreloadInstr()

This procedure performs a *preload instructions* hint.

```
Hint_PreloadInstr(bits(32) address)
```

G.7.25 Hint_SendEvent()

This procedure performs a *send event* hint.

G.7.26 Hint_Yield()

This procedure performs a *Yield* hint.

G.7.27 InstructionSynchronizationBarrier()

This procedure produces an Instruction Synchronization Barrier.

```
InstructionSynchronizationBarrier(bits(4) option)
```

G.7.28 IntegerZeroDivideTrappingEnabled()

This function returns TRUE if the trapping of divisions by zero in the integer division instructions `SDIV` and `UDIV` is enabled, and FALSE otherwise.

In the M profile, this is controlled by the DIV_0_TRP bit in the Configuration Control register. TRUE is returned if the bit is 1 and FALSE if it is 0.

G.7.29 ProcessorID()

Identifies the executing processor.

G.7.30 SetPending()

This procedure sets the associated exception state to Pending. For a definition of the different exception states see *Exceptions* on page B1-5.

G.7.31 ThisInstr()

This function returns the currently-executing instruction. It is only used on 32-bit instruction encodings at present.

```
bits(32) ThisInstr()
```

Appendix H
Pseudocode Index

This appendix provides an index to pseudocode operators and functions that occur elsewhere in the document. It contains the following sections:

H.1 Pseudocode operators and keywords

Table H-1 lists the pseudocode operators and keywords, and is an index to their descriptions:

Table H-1 Pseudocode operators and keywords

Operator	Meaning	See
-	Unary minus on integers or reals	*Unary plus, minus and absolute value* on page AppxG-15
-	Subtraction of integers, reals and bitstrings	*Addition and subtraction* on page AppxG-15
+	Unary plus on integers or reals	*Unary plus, minus and absolute value* on page AppxG-15
+	Addition of integers, reals and bitstrings	*Addition and subtraction* on page AppxG-15
(...)	Around arguments of procedure	*Procedure calls* on page AppxG-17, *Procedure and function definitions* on page AppxG-21
(...)	Around arguments of function	*General expression syntax* on page AppxG-9, *Procedure and function definitions* on page AppxG-21
.	Extract named member from a list	*Lists* on page AppxG-7
.	Extract named bit or field from a register	*Bitstring extraction* on page AppxG-12
!	Boolean NOT	*Operations on booleans* on page AppxG-11
!=	Compare for non-equality (any type)	*Equality and non-equality testing* on page AppxG-11
!=	Compare for non-equality (between integers and reals)	*Comparisons* on page AppxG-15
&&	Boolean AND	*Operations on booleans* on page AppxG-11
*	Multiplication of integers and reals	*Multiplication* on page AppxG-15
/	Division of integers and reals (real result)	*Division and modulo* on page AppxG-15
/*...*/	Comment delimiters	*Comments* on page AppxG-21
//	Introduces comment terminated by end of line	*Comments* on page AppxG-21
:	Bitstring concatenation	*Bitstring concatenation and replication* on page AppxG-12
:	Integer range in bitstring extraction operator	*Bitstring extraction* on page AppxG-12
[...]	Around array index	*Arrays* on page AppxG-8

Table H-1 Pseudocode operators and keywords (continued)

Operator	Meaning	See
[...]	Around arguments of array-like function	*General expression syntax* on page AppxG-9, *Procedure and function definitions* on page AppxG-21
^	Boolean exclusive-OR	*Operations on booleans* on page AppxG-11
||	Boolean OR	*Operations on booleans* on page AppxG-11
<	*Less than* comparison of integers and reals	*Comparisons* on page AppxG-15
<...>	Extraction of specified bits of bitstring or integer	*Bitstring extraction* on page AppxG-12
<<	Multiply integer by power of 2 (with rounding towards -infinity)	*Scaling* on page AppxG-16
<=	*Less than or equal* comparison of integers and reals	*Comparisons* on page AppxG-15
=	Assignment	*Assignments* on page AppxG-17
==	Compare for equality (any type)	*Equality and non-equality testing* on page AppxG-11
==	Compare for equality (between integers and reals)	*Comparisons* on page AppxG-15
>	*Greater than* comparison of integers and reals	*Comparisons* on page AppxG-15
>=	*Greater than or equal* comparison of integers and reals	*Comparisons* on page AppxG-15
>>	Divide integer by power of 2 (with rounding towards -infinity)	*Scaling* on page AppxG-16
2^N	Power of two (real result)	*Scaling* on page AppxG-16
AND	Bitwise AND of bitstrings	*Logical operations on bitstrings* on page AppxG-13
array	Keyword introducing array type definition	*Arrays* on page AppxG-8
bit	Bitstring type of length 1	*Bitstrings* on page AppxG-5
bits(N)	Bitstring type of length N	*Bitstrings* on page AppxG-5
boolean	Boolean type	*Booleans* on page AppxG-6
case ... of ...	Control structure	*case ... of ...* on page AppxG-20
DIV	Quotient from integer division	*Division and modulo* on page AppxG-15
enumeration	Keyword introducing enumeration type definition	*Enumerations* on page AppxG-6

Table H-1 Pseudocode operators and keywords (continued)

Operator	Meaning	See
`EOR`	Bitwise EOR of bitstrings	*Logical operations on bitstrings* on page AppxG-13
`FALSE`	Boolean constant	*Booleans* on page AppxG-6
`for ...`	Control structure	*for ...* on page AppxG-20
`if ... then ... else ...`	Expression selecting between two values	*Conditional selection* on page AppxG-11
`if ... then ... else ...`	Control structure	*if ... then ... else ...* on page AppxG-18
`IMPLEMENTATION_DEFINED`	Describes IMPLEMENTATION DEFINED behavior	*IMPLEMENTATION_DEFINED* on page AppxG-18
`integer`	Unbounded integer type	*Integers* on page AppxG-6
`MOD`	Remainder from integer division	*Division and modulo* on page AppxG-15
`OR`	Bitwise OR of bitstrings	*Logical operations on bitstrings* on page AppxG-13
`otherwise`	Introduces default case in `case ... of ...` control structure	*case ... of ...* on page AppxG-20
`real`	Real number type	*Reals* on page AppxG-6
`repeat ... until ...`	Control structure	*repeat ... until ...* on page AppxG-19
`return`	Procedure or function return	*Return statements* on page AppxG-17
`SEE`	Points to other pseudocode to use instead	*SEE...* on page AppxG-18
`SUBARCHITECTURE_DEFINED`	Describes SUBARCHITECTURE DEFINED behavior	*SUBARCHITECTURE_DEFINED* on page AppxG-18
`TRUE`	Boolean constant	*Booleans* on page AppxG-6
`UNDEFINED`	Cause Undefined Instruction exception	*UNDEFINED* on page AppxG-17
`UNKNOWN`	Unspecified value	*General expression syntax* on page AppxG-9
`UNPREDICTABLE`	Unspecified behavior	*UNPREDICTABLE* on page AppxG-17
`when`	Introduces specific case in `case ... of ...` control structure	*case ... of ...* on page AppxG-20
`while ... do ...`	Control structure	*while ... do* on page AppxG-20

H.2 Pseudocode functions and procedures

Table H-2 lists the pseudocode functions and procedures used in this manual, and is an index to their descriptions:

Table H-2 Pseudocode functions and procedures

Function	Meaning	See
`_Mem[]`	Basic memory accesses	*Basic memory accesses* on page B2-4
`Abs()`	Absolute value of an integer or real	*Unary plus, minus and absolute value* on page AppxG-15
`AddWithCarry()`	Addition of bitstrings, with carry input and carry/overflow outputs	*Pseudocode details of addition and subtraction* on page A2-8
`Align()`	Align integer or bitstring to multiple of an integer	*Rounding and aligning* on page AppxG-16
`ALUWritePC()`	Write value to PC, with interworking for ARM only from ARMv7	*Pseudocode details of ARM core register operations* on page A2-11
`ArchVersion()`	Major version number of the architecture	*ArchVersion()* on page AppxG-22
`ASR()`	Arithmetic shift right of a bitstring	*Shift and rotate operations* on page A2-5
`ASR_C()`	Arithmetic shift right of a bitstring, with carry output	*Shift and rotate operations* on page A2-5
`BadReg()`	Test for register number 13 or 15	*BadReg()* on page AppxG-22
`BigEndianReverse()`	Endian-reverse the bytes of a bitstring	*Reverse endianness* on page B2-7
`BitCount()`	Count number of ones in a bitstring	*Bitstring count* on page AppxG-13
`BranchTo()`	Continue execution at specified address	*Pseudocode details for ARM core register access in the Thumb instruction set* on page B1-12
`BranchWritePC()`	Write value to PC, without interworking	*Pseudocode details of ARM core register operations* on page A2-11
`BXWritePC()`	Write value to PC, with interworking	
`CallSupervisor()`	Generate exception for SVC instruction	*CallSupervisor()* on page AppxG-22
`CheckPermission()`	Memory system check of access permission	*Access permission checking* on page B2-10
`ClearEventRegister()`	Clear the Event Register of the current processor	*Pseudocode details of the Wait For Event lock mechanism* on page B1-50
`ClearExclusiveByAddress()`	Clear local exclusive monitor records for an address range	*Pseudocode details of operations on exclusive monitors* on page B2-8
`ClearExclusiveLocal()`	Clear global exclusive monitor record for a processor	
`ConditionPassed()`	Returns TRUE if the current instruction passes its condition check	*Pseudocode details of conditional execution* on page A6-9

Table H-2 Pseudocode functions and procedures (continued)

Function	Meaning	See
`Consistent()`	Test identically-named instruction bits or fields are identical	*Instruction encoding diagrams and pseudocode* on page AppxG-2
`Coproc_Accepted()`	Determine whether a coprocessor accepts an instruction.	*Coproc_Accepted()* on page AppxG-22
`Coproc_DoneLoading()`	Returns TRUE if enough words have been loaded, for an LDC or LDC2 instruction	*Coproc_DoneLoading()* on page AppxG-22
`Coproc_DoneStoring()`	Returns TRUE if enough words have been stored, for an STC or STC2 instruction	*Coproc_DoneStoring()* on page AppxG-23
`Coproc_GetOneWord()`	Get word from coprocessor, for an MRC or MRC2 instruction	*Coproc_GetOneWord()* on page AppxG-23
`Coproc_GetTwoWords()`	Get two words from coprocessor, for an MRRC or MRRC2 instruction	*Coproc_GetTwoWords()* on page AppxG-23
`Coproc_GetWordToStore()`	Get next word to store from coprocessor, for STC or STC2 instruction	*Coproc_GetWordToStore()* on page AppxG-23
`Coproc_InternalOperation()`	Instruct coprocessor to perform an internal operation, for a CDP or CDP2 instruction	*Coproc_InternalOperation()* on page AppxG-23
`Coproc_SendLoadedWord()`	Send next loaded word to coprocessor, for LDC or LDC2 instruction	*Coproc_SendLoadedWord()* on page AppxG-23
`Coproc_SendOneWord()`	Send word to coprocessor, for an MCR or MCR2 instruction	*Coproc_SendOneWord()* on page AppxG-23
`Coproc_SendTwoWords()`	Send two words to coprocessor, for an MCRR or MCRR2 instruction	*Coproc_SendTwoWords()* on page AppxG-23
`CountLeadingSignBits()`	Number of identical sign bits at left end of bitstring, excluding the leftmost bit itself	*Lowest and highest set bits of a bitstring* on page AppxG-13
`CountLeadingZeroBits()`	Number of zeros at left end of bitstring	
`CurrentCond()`	Returns condition for current instruction	*Pseudocode details of conditional execution* on page A6-9
`DataAddressMatch()`	DWT comparator data address matching	*Comparator support - data address matching* on page C1-44
`DataMemoryBarrier()`	Perform a Data Memory Barrier operation	*DataMemoryBarrier()* on page AppxG-24
`DataSynchronizationBarrier()`	Perform a Data Synchronization Barrier operation	*DataSynchronizationBarrier()* on page AppxG-24
`Deactivate()`	Removal of Active state from an exception as part of the exception return	*Exception return behavior* on page B1-25
`DecodeImmShift()`	Decode shift type and amount for an immediate shift	*Shift operations* on page A6-13

Table H-2 Pseudocode functions and procedures (continued)

Function	Meaning	See
`DecodeRegShift()`	Decode shift type for a register-controlled shift	*Shift operations* on page A6-13
`DefaultTEXDecode()`	Determine memory attributes for a set of TEX[2:0], C, B bits	*MPU access control decode* on page B2-10
~~`DefaultAttributes`~~`()`	Determine memory attributes for an address in the default memory map	*Default memory access decode* on page B2-11
`EncodingSpecificOperations()`	Invoke encoding-specific pseudocode and *should be* checks	*Instruction encoding diagrams and pseudocode* on page AppxG-2
`EventRegistered()`	Determine whether the Event Register of the current processor is set	*Pseudocode details of the Wait For Event lock mechanism* on page B1-50
`ExceptionEntry()`	Exception entry behavior	*Exception entry behavior* on page B1-21
`ExceptionIN()`	Determine exception entry status	*External interrupt input behavior* on page B3-29
`ExceptionOUT()`	Determine exception return status	*External interrupt input behavior* on page B3-29
`ExceptionReturn()`	Exception return behavior	*Exception return behavior* on page B1-25
`ExceptionTaken()`	Part of `ExceptionEntry()` behavior	*Exception entry behavior* on page B1-21
`ExclusiveMonitorsPass()`	Check whether Store-Exclusive operation has control of exclusive monitors	*Pseudocode details of operations on exclusive monitors* on page B2-8
`FindPriv()`	Determine access privilege	*Interfaces to memory system specific pseudocode* on page B2-4
`HighestSetBit()`	Position of leftmost 1 in a bitstring	*Lowest and highest set bits of a bitstring* on page AppxG-13
`Hint_Debug()`	Perform function of DBG hint instruction	*Hint_Debug()* on page AppxG-24
`Hint_PreloadData()`	Perform function of PLD memory hint instruction	*Hint_PreloadData()* on page AppxG-24
~~`Hint_PreloadData()`~~	~~Perform function of PLDW memory hint instruction~~	~~*Hint_PreloadDataForWrite()* on page AppxG-24~~
`Hint_PreloadInstr()`	Perform function of PLI memory hint instruction	*Hint_PreloadInstr()* on page AppxG-25
`Hint_SendEvent()`	Perform function of SEV hint instruction	*Hint_SendEvent()* on page AppxG-25
`Hint_Yield()`	Perform function of YIELD hint instruction	*Hint_Yield()* on page AppxG-25
`GenerateCoprocessorException()`	Generate the exception for an unclaimed coprocessor instruction	*GenerateCoprocessorException()* on page AppxG-24

Table H-2 Pseudocode functions and procedures (continued)

Function	Meaning	See
`GenerateIntegerZeroDivide()`	Generate the exception for a trapped divide-by-zero on execution of an integer divide instruction	*GenerateIntegerZeroDivide()* on page AppxG-24
`ITAdvance()`	Advance the ITSTATE bits to their values for the next instruction	*Pseudocode details of ITSTATE operation* on page A6-11,
`InITBlock()`	Return TRUE if current instruction is in an IT block.	
`InterruptAssertion()`	Determine status of an external interrupt	*External interrupt input behavior* on page B3-29
`InstrAddressMatch()`	DWT comparator instruction address matching	*Comparator support - instruction address matching* on page C1-43
`InstructionSynchronizationBarrier()`	Perform an Instruction Synchronization Barrier operation	*InstructionSynchronizationBarrier()* on page AppxG-25
`Int()`	Convert bitstring to integer in argument-specified fashion	*Converting bitstrings to integers* on page AppxG-14
`IntegerZeroDivideTrappingEnabled()`	Check whether divide-by-zero trapping is enabled for integer divide instructions	*IntegerZeroDivideTrappingEnabled()* on page AppxG-25
`IsExclusiveGlobal()`	Check a global exclusive access record	on page B2-8*Pseudocode details of operations on exclusive monitors* on page B2-8
`IsExclusivelocal()`	Check a local exclusive access record	
`IsOnes()`	Test for all-ones bitstring (Boolean result)	*Testing a bitstring for being all zero or all ones* on page AppxG-13
`IsOnesBit()`	Test for all-ones bitstring (bit result)	
`IsZero()`	Test for all-zeros bitstring (Boolean result)	*Testing a bitstring for being all zero or all ones* on page AppxG-13
`IsZeroBit()`	Test for all-zeros bitstring (bit result)	
`LastInITBlock()`	Return TRUE if current instruction is the last instruction in an IT block.	*Pseudocode details of ITSTATE operation* on page A6-11
`LateArrival()`	Late arrival exception handling	*Late arriving exceptions* on page B1-33
`Len()`	Bitstring length	*Bitstring length and top bit* on page AppxG-12
`LoadWritePC()`	Write value to PC, with interworking (without it before ARMv5T)	*Pseudocode details of ARM core register operations* on page A2-11
`LookUpSP()`	Select the current SP	*Pseudocode details for ARM core register access in the Thumb instruction set* on page B1-12
`LowestSetBit()`	Position of rightmost 1 in a bitstring	*Lowest and highest set bits of a bitstring* on page AppxG-13

Table H-2 Pseudocode functions and procedures (continued)

Function	Meaning	See
`LSL()`	Logical shift left of a bitstring	*Shift and rotate operations* on page A2-5
`LSL_C()`	Logical shift left of a bitstring, with carry output	
`LSR()`	Logical shift right of a bitstring	
`LSR_C()`	Logical shift right of a bitstring, with carry output	
`MarkExclusiveGlobal()`	Set a global exclusive access record	on page B2-8*Pseudocode details of operations on exclusive monitors* on page B2-8
~~`IMarkExclusivelocal()`~~	Set a local exclusive access record	
`Max()`	Maximum of integers or reals	*Maximum and minimum* on page AppxG-16
`MemA[]`	Memory access that must be aligned, at current privilege level	*Aligned memory accesses* on page B2-5
`MemA_unpriv[]`	Memory access that must be aligned, unprivileged	
`MemA_with_priv[]`	Memory access that must be aligned, at specified privilege level	
`MemU[]`	Memory access without alignment requirement, at current privilege level	*Unaligned memory accesses* on page B2-6
`MemU_unpriv[]`	Memory access without alignment requirement, unprivileged	
`MemU_with_priv[]`	Memory access without alignment requirement, at specified privilege level	
`Min()`	Minimum of integers or reals	*Maximum and minimum* on page AppxG-16
`NOT()`	Bitwise inversion of a bitstring	*Logical operations on bitstrings* on page AppxG-13
`Ones()`	All-ones bitstring	*Bitstring concatenation and replication* on page AppxG-12
`ProcessorID()`	Return integer identifying the processor	*ProcessorID()* on page AppxG-25
`PopStack()`	Stack restore sequence on an exception return	*Exception return behavior* on page B1-25
`PushStack()`	Stack save sequence on exception entry	*Exception entry behavior* on page B1-21

Table H-2 Pseudocode functions and procedures (continued)

Function	Meaning	See
`R[]`	Access the main ARM core register bank	*Pseudocode details of ARM core register operations* on page A2-11 *Pseudocode details for ARM core register access in the Thumb instruction set* on page B1-12
`Replicate()`	Bitstring replication	*Bitstring concatenation and replication* on page AppxG-12
`ReturnAddress()`	Return address stacked on exception entry	*Exception entry behavior* on page B1-21
`ROR()`	Rotate right of a bitstring	*Shift and rotate operations* on page A2-5
`ROR_C()`	Rotate right of a bitstring, with carry output	
`RRX()`	Rotate right with extend of a bitstring	
`RRX_C()`	Rotate right with extend of a bitstring, with carry output	
`Sat()`	Convert integer to bitstring with specified saturation	*Pseudocode details of saturation* on page A2-9
`SatQ()`	Convert integer to bitstring with specified saturation, with saturated flag output	
`SendEvent()`	Create a WFE wake up event that sets the Event Register(s) on execution of an `SEV` instruction.	*Pseudocode details of the Wait For Event lock mechanism* on page B1-50
`SetEventRegister()`	Set the Event Register of the current processor	*Pseudocode details of the Wait For Event lock mechanism* on page B1-50
`SetExclusiveMonitors()`	Set exclusive monitors for a local exclusive operation	*Pseudocode details of operations on exclusive monitors* on page B2-8
`Shift()`	Perform a specified shift by a specified amount on a bitstring	*Shift operations* on page A6-13
`Shift_C()`	Perform a specified shift by a specified amount on a bitstring, with carry output	
`SignedSat()`	Convert integer to bitstring with signed saturation	*Pseudocode details of saturation* on page A2-9
`SignedSatQ()`	Convert integer to bitstring with signed saturation, with saturated flag output	
`SignExtend()`	Extend bitstring to left with copies of its leftmost bit	*Zero-extension and sign-extension of bitstrings* on page AppxG-13
`SInt()`	Convert bitstring to integer in signed (two's complement) fashion	*Converting bitstrings to integers* on page AppxG-14
`TailChain()`	Tail chaining exception behavior	*Tail-chaining* on page B1-36

Table H-2 Pseudocode functions and procedures (continued)

Function	Meaning	See
`TakeReset()`	Reset behavior	*Reset behavior* on page B1-20
`ThisInstr()`	Returns the bitstring encoding of the current instruction	*ThisInstr()* on page AppxG-25
`ThumbExpandImm()`	Expansion of immediates for Thumb instructions	*Operation* on page A5-16
`ThumbExpandImmWithC()`	Expansion of immediates for Thumb instructions, with carry output	
`TopBit()`	Leftmost bit of a bitstring	*Bitstring length and top bit* on page AppxG-12
`UInt()`	Convert bitstring to integer in unsigned fashion	*Converting bitstrings to integers* on page AppxG-14
`UnsignedSat()`	Convert integer to bitstring with unsigned saturation	*Pseudocode details of saturation* on page A2-9
`UnsignedSatQ()`	Convert integer to bitstring with unsigned saturation, with saturated flag output	
`ValidateAddress()`	Resolve the permissions and memory attributes for a PMSA memory access	*MPU pseudocode* on page B3-36
`WaitForEvent()`	Wait until `WFE` instruction completes	*Pseudocode details of the Wait For Event lock mechanism* on page B1-50
`WaitForInterrupt()`	Wait until `WFI` instruction completes	*Pseudocode details of Wait For Interrupt* on page B1-52
`WriteToRegField()`	Indicate a write of ’1’ to a specified field in a system control register	*External interrupt input behavior* on page B3-29
`ZeroExtend()`	Extend bitstring to left with zero bits	*Zero-extension and sign-extension of bitstrings* on page AppxG-13
`Zeros()`	All-zeros bitstring	*Bitstring concatenation and replication* on page AppxG-12

Appendix I
Register Index

This appendix provides an index to the descriptions of the ARM registers (core and memory mapped) in the document. It contains the following sections:

I.1 ARM core registers

Table I-1 provides an index to the main descriptions of the ARM core registers defined in ARMv7-M.

Table I-1 ARM core register index

Register	Description, see
R0, R1, R2, R3, R4, R5, R6, R7, R8, R9, R10, R11, R12	*Registers* on page B1-8
SP_main, SP_process	*The SP registers* on page B1-8
LR (R14)	*Registers* on page B1-8
PC (R15)	*Registers* on page B1-8
APSR[a]	*The special-purpose program status registers (xPSR)* on page B1-8
IPSR[a]	*The special-purpose program status registers (xPSR)* on page B1-8
EPSR[a]	*The special-purpose program status registers (xPSR)* on page B1-8
PRIMASK	*The special-purpose mask registers* on page B1-10
FAULTMASK	*The special-purpose mask registers* on page B1-10
BASEPRI	*The special-purpose mask registers* on page B1-10
CONTROL	*The special-purpose control register* on page B1-11

[a] xPSR = APSR | IPSR | EPSR

I.2 Memory mapped system registers

Table I-2 provides an index to the main descriptions of the memory mapped system control registers defined in ARMv7-M. The registers are listed in the order they are described in this manual.

Table I-2 Memory-mapped control register index

Register	Description, see
ACTLR	*Auxiliary Control Register – (0xE000E008)* on page B3-9
CPUID	*CPUID Base Register – (CPUID, 0xE000ED00)* on page B3-10
ICSR	*Interrupt Control and State Register – (0xE000ED04)* on page B3-12
VTOR	*Vector Table Offset Register – (0xE000ED08)* on page B3-13
AIRCR	*Application Interrupt and Reset Control Register – (0xE000ED0C)* on page B3-14
SCR	*System Control Register (0xE000ED10)* on page B3-15
CCR	*Configuration and Control Register (0xE000ED14)* on page B3-16
SHPR1	*System Handler Priority Register 1 – (0xE000ED18)* on page B3-17
SHPR2	*System Handler Priority Register 2 – (0xE000ED1C)* on page B3-17
SHPR3	*System Handler Priority Register 3 – (0xE000ED20)* on page B3-17
SHCSR	*System Handler Control and State Register – (0xE000ED24)* on page B3-18
CFSR	*Configurable Fault Status Registers (CFSR, 0xE000ED28)* on page B3-19
MMSR	*MemManage Status Register (MMFSR, 0xE000D28)* on page B3-19
BFSR	*BusFault Status Register (BFSR, 0xE000ED29)* on page B3-20
UFSR	*UsageFault Status Register (UFSR, 0xE000ED2A)* on page B3-20
HFSR	*HardFault Status Register (0xE000ED2C)* on page B3-21
MMAR	*MemManage Address Register (0xE000ED34)* on page B3-22
BFAR	*BusFault Address Register (0xE000ED38)* on page B3-22
~~CAR~~	*Coprocessor Access Control Register– (0xE000ED88)* on page B3-22
STIR	*Software Trigger Interrupt Register – (0xE000EF00)* on page B3-23
~~STCSR~~	*SysTick Control and Status Register – (0xE000E010)* on page B3-26
~~STRVR~~	*SysTick Reload Value Register – (0xE000E014)* on page B3-26

Table I-2 Memory-mapped control register index (continued)

Register	Description, see
~~STCVR~~	*SysTick Current Value Register – (0xE000E018)* on page B3-27
~~STCR~~	*SysTick Calibration Value Register – (0xE000E01C)* on page B3-27
ICTR	*Interrupt Controller Type Register – (0xE000E004)* on page B3-32
~~ISER[x]~~	*Interrupt Set-Enable Registers – (0xE000E100-E17C)* on page B3-33
~~ICER[x]~~	*Interrupt Clear-Enable Registers – (0xE000E180-E1FC)* on page B3-33
~~ISPR[x]~~	*Interrupt Set-Pending Registers – (0xE000E200-E27C)* on page B3-33
~~ICPR[x]~~	*Interrupt Clear-Pending Registers – (0xE000E280-E2FC)* on page B3-34
~~IABR[x]~~	*Interrupt Active Bit Registers – (0xE000E300-E37C)* on page B3-34
~~IPR[x]~~	*Interrupt Priority Registers – (0xE000E400-E7F8)* on page B3-34
~~MPUTR~~	*MPU Type Register – (0xE000ED90)* on page B3-39
~~MPUCR~~	*MPU Control Register – (0xE000ED94)* on page B3-40
~~MPURNR~~	*MPU Region Number Register – (0xE000ED98)* on page B3-41
~~MPURBAR~~	*MPU Region Base Address Register – (0xE000ED9C)* on page B3-41
~~MPURASR~~	*MPU Region Attribute and Size Register – (0xE000EDA0)* on page B3-42

I.3 Memory mapped debug registers

Table I-3 provides an index to the main descriptions of the memory mapped debug registers defined in the ARMv7-M Debug Extension. The registers are listed in the order they are described in this manual.

Table I-3 Memory-mapped debug register index

Register[a]	Description, see
General infrastructure and CoreSight registers	For general infrastructure and CoreSight register support, see Table C1-2 on page C1-4, Appendix B *ARMv7-M infrastructure IDs* and *CoreSight® Architecture Specification*.
DFSR	*Debug Fault Status Register (0xE000ED30)* on page C1-19
DHCSR	*Debug Halting Control and Status Register – (0xE000EDF0)* on page C1-20
DCRSR	*Debug Core Register Selector Register – (0xE000EDF4)* on page C1-22
DCRDR	*Debug Core Register Data Register (DCRDR)* on page C1-23
DEMCR	*Debug Exception and Monitor Control Register – (0xE000EDFC)* on page C1-24
~~STIMx (ITM support)~~	*Stimulus Portx Register - ITM_STIM[255:0] (0xE0000000 to 0xE00003FC)* on page C1-30
~~TERx (ITM support)~~	*Trace Enable Register - ITM_TER[7:0] (0xE0000E00 to 0xE0000E1C)* on page C1-30
~~TPR (ITM support)~~	*Trace Privilege Register - ITM_TPR (0xE0000E40)* on page C1-31
~~TCR (ITM support)~~	*Trace Control Register – ITM_TCR (0xE0000E80)* on page C1-31
DWT_CTRL	*Control Register (DWT_CTRL)* on page C1-48
DWT_CYCCNT	*Cycle Count Register (DWT_CYCCNT)* on page C1-49
DWT_CPICNT	*CPI Count Register (DWT_CPICNT)* on page C1-50
DWT_EXCCNT	*Exception Overhead Count Register (DWT_EXCCNT)* on page C1-50
DWT_SLEEPCNT	*Sleep Count Register (DWT_SLEEPCNT)* on page C1-51
DWT_LSUCNT	*LSU Count Register (DWT_LSUCNT)* on page C1-51
DWT_FOLDCNT	*Folded-instruction Count Register (DWT_FOLDCNT)* on page C1-52
DWT_PCSR	*Program Counter Sample Register (DWT_PCSR)* on page C1-52
DWT_COMPx	*Comparator Register (DWT_COMPx)* on page C1-53
DWT_MASKx	*Mask Register (DWT_MASKx)* on page C1-53
DWT_FUNCTIONx	*Function Register (DWT_FUNCTIONx)* on page C1-54

Table I-3 Memory-mapped debug register index (continued)

Register[a]	Description, see
ETM registers	For ETM related registers, see *Embedded Trace Macrocell Architecture Specification*.
TPIU_SSPSR	*Supported Synchronous Port Sizes Register (TPIU_SSPSR, 0xE0040000)* on page C1-58
TPIU_CSPSR	*Current Synchronous Port Size Register (TPIU_CSPSR, 0xE0040004)* on page C1-58
TPIU_ACPR	*Asynchronous Clock Prescaler Register (TPIU_ACPR, 0xE0040010)* on page C1-59
TPIU_SPPR	*Selected Pin Protocol Register (TPIU_SPPR, 0xE00400F0)* on page C1-59
~~TPIU_TR~~	*TPIU Type Register (TPIU_TYPE, 0xE0040FC8)* on page C1-60
FP_CTRL	*FlashPatch Control Register (FP_CTRL)* on page C1-64
FP_REMAP	*FlashPatch Remap Register (FP_REMAP)* on page C1-64
FP_COMPx	*FlashPatch Comparator Register – instruction comparison (FP_COMPx)* on page C1-65 and *FlashPatch Comparator Register – literal comparison (FP_COMPx)* on page C1-66

[a] In addition to the registers listed, debug support includes bits in the ICSR, see *Interrupt Control State Register (ICSR)* on page B3-12.

Glossary

AAPCS

Procedure Call Standard for the ARM Architecture.

Addressing mode

Means a method for generating the memory address used by a load/store instruction.

Aligned

Refers to data items stored in such a way that their address is divisible by the highest power of 2 that divides their size. Aligned halfwords, words and doublewords therefore have addresses that are divisible by 2, 4 and 8 respectively.

An aligned access is one where the address of the access is aligned to the size of an element of the access

APSR

See Application Program Status Register.

Application Program Status Register

The register containing those bits that deliver status information about the results of instructions, the N, Z, C, and V bits of the xPSR. See *The special-purpose program status registers (xPSR)* on page B1-8.

Atomicity

Is a term that describes either single-copy atomicity or multi-copy atomicity. The forms of atomicity used in the ARM architecture are defined in *Atomicity in the ARM architecture* on page A3-20.

See also Multi-copy Atomicity, Single-copy atomicity.

Banked register

Is a register that has multiple instances, with the instance that is in use depending on the processor mode, security state, or other processor state.

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 that is sent to memory.

Base register write-back

Describes writing back a modified value to the base register used in an address calculation.

Big-endian memory

Means that:

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

Blocking

Describes an operation that does not permit following instructions to be executed before the operation is completed.

A non-blocking operation can permit following instructions to be executed before the operation is completed, and in the event of encountering an exception do not signal an exception to the core. This enables implementations to retire following instructions while the non-blocking operation is executing, without the need to retain precise processor state.

Branch prediction

Is where a processor chooses a future execution path to prefetch along (see Prefetching). For example, after a branch instruction, the processor can choose to prefetch either the instruction following the branch or the instruction at the branch target.

Breakpoint

Is a debug event triggered by the execution of a particular instruction, specified in terms of the address of the instruction and/or the state of the processor when the instruction is executed.

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 that 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 must be aligned to a suitable memory boundary. A *memory cache line* is a block of memory locations with the same size and alignment as a cache line. Memory cache lines are sometimes loosely just called cache lines.

Cache miss

Is a memory access that cannot be processed at high speed because the data it addresses is not in the cache.

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.

Clear Relates to registers or register fields. Indicates the bit has a value of zero (or bit field all 0s), or is being written with zero or all 0s.

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.

Configuration

Settings made on reset, or immediately after reset, and normally expected to remain static throughout program execution.

Context switch

Is the saving and restoring of computational state when switching between different threads or processes. In this manual, the term context switch is used to describe any situations where the context is switched by an operating system and might or might not include changes to the address space.

DCB Debug Control Block - a region within the System Control Space (see SCS) specifically assigned to register support of debug features.

Digital signal processing (DSP)

Refers to a variety of algorithms that are used to process signals that have been sampled and converted to digital form. Saturated arithmetic is often used in such algorithms.

Direct Memory Access

Is an operation that accesses main memory directly, without the processor performing any accesses to the data concerned.

Do-not-modify fields (DNM)

Means the value must not be altered by software. DNM fields read as UNKNOWN values, and can only be written with the same value read from the same field on the same processor.

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

DWT Data Watchpoint and Trace - part of the ARM debug architecture.

Endianness

Is an aspect of the system's memory mapping. See big-endian and little-endian.

EPSR *See* Execution Program Status Register.

ETM Embedded Trace Macrocell - part of the ARM debug architecture

Exception

Handles an event. For example, an exception could handle an external interrupt or an ~~Undefined~~ Instruction.

Exception vector

Is one of a number of fixed addresses in low memory, or in high memory if high vectors are configured.

Execution Program Status Register

The register that contains the execution state bits and is part of the xPSR. See *The special-purpose program status registers (xPSR)* on page B1-8.

Execution stream

The stream of instructions that would have been executed by sequential execution of the program.

Explicit access

A read from memory, or a write to memory, generated by a load or store instruction executed in the CPU. Reads and writes generated by L1 DMA accesses or hardware translation table accesses are not explicit accesses.

Fault An exception due to some form of system error.

General-purpose register

Is one of the 32-bit general-purpose integer registers, R0 to R15. Note that R15 holds the Program Counter, and there are often limitations on its use that do not apply to R0 to R14.

Halfword

Is a 16-bit data item. Halfwords are normally halfword-aligned in ARM systems.

Halfword-aligned

Means that the address is divisible by 2.

High registers

Are ARM core registers 8 to 15, that can be accessed by some Thumb instructions.

Immediate and offset fields

Are unsigned unless otherwise stated.

Immediate values

Are values that are encoded directly in the instruction and used as numeric data when the instruction is executed. Many ARM and Thumb instructions permit small numeric values to be encoded as immediate values in 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

Means that the behavior is not architecturally defined, but should be defined and documented by individual implementations.

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 address that is sent to memory. Some addressing modes optionally permit the index register value to be shifted before 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.

Interrupt Program Status Register

The register that provides status information on whether an application thread or exception handler is currently executing on the processor. If an exception handler is executing, the register provides information on the exception type. The register is part of the xPSR. See *The special-purpose program status registers (xPSR)* on page B1-8.

Interworking

Is a method of working that permits branches between ARM and Thumb code in architecture variants that support both execution states.

IPSR *See* Interrupt Program Status Register.

IT block An IT block is a block of up to four instructions following an *If-Then* (IT) instruction. Each instruction in the block is conditional. The conditions for the instructions are either all the same, or some can be the inverse of others. See *IT* on page A6-78 for additional information.

ITM Instrumentation Trace Macrocell - part of the ARM debug architecture

Little-endian memory

Means that:

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

Load/Store architecture

Is an architecture where data-processing operations only operate on register contents, not directly on memory contents.

Long branch

Is the use of a load instruction to branch to anywhere in the 4GB address space.

Memory barrier

See *Memory barriers* on page A3-35.

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 hint

A memory hint instruction allows you to provide advance information to memory systems about future memory accesses, without actually loading or storing any data to or from the register file. `PLD` and `PLI` are the only memory hint instructions defined in ARMv7-M.

Memory-mapped I/O

Uses special memory addresses that supply I/O functions when they are loaded from or stored to.

Memory Protection Unit (MPU)

Is a hardware unit whose registers provide simple control of a limited number of protection regions in memory.

MPU *See* Memory Protection Unit.

NRZ Non-Return-to-Zero - physical layer signalling scheme used on asynchronous communication ports.

Multi-copy atomicity

Is the form of atomicity described in *Multi-copy atomicity* on page A3-21.

See also Atomicity, Single-copy atomicity.

Offset addressing

Means that the memory address is formed by adding or subtracting an offset to or from the base register value.

Physical address

Identifies a main memory location.

Post-indexed addressing

Means that the memory address is the base register value, but an offset is added to or subtracted from the base register value and the result is written back to the base register.

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 access

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.

Protection region

Is a memory region whose position, size, and other properties are defined by Memory Protection Unit registers.

Protection Unit

See Memory Protection Unit.

Pseudo-instruction

UAL assembler syntax that assembles to an instruction encoding that is expected to disassemble to a different assembler syntax, and is described in this manual under that other syntax. For example, `MOV <Rd>,<Rm>, LSL #<n>` is a pseudo-instruction that is expected to disassemble as `LSL <Rd>,<Rm>,#<n>`

PSR Program Status Register. *See* APSR, EPSR, IPSR and xPSR.

RAZ *See* Read-As-Zero fields.

RAO/SBOP field

Read-As-One, Should-Be-One-or-Preserved on writes.

In any implementation, the bit must read as 1 (or all 1s for a bit field), and writes to the field must be ignored.

Software can rely on the field reading as 1 (or all 1s), but must use an SBOP policy to write to the field.

RAZ/SBZP field

Read-As-Zero, Should-Be-Zero-or-Preserved on writes.

In any implementation, the bit must read as 0 (or all 0s for a bit field), and writes to the field must be ignored.

Software can rely on the field reading as zero, but must use an SBZP policy to write to the field.

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.

Reserved

Unless otherwise stated:

- instructions that are reserved or that access reserved registers have UNPREDICTABLE behavior
- bit positions described as Reserved are UNK/SBZP.

Return Link

a value relating to the return address

R/W1C register bits marked R/W1C can be read normally and support write-one-to-clear. A read then write of the result back to the register will clear all bits set. R/W1C protects against read-modify-write errors occurring on bits set between reading the register and writing the value back (since they are written as zero, they will not be cleared).

RAZ/WI Relates to registers or register fields. Read as zero, ignore writes. RAZ can be used on its own.

RO Read only register or register field. RO bits are ignored on write accesses.

RISC Reduced Instruction Set Computer.

RMW *See* Read-Modify-Write fields.

Rounding error
Is defined to be the value of the rounded result of an arithmetic operation minus the exact result of the operation.

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.

SBO *See* Should-Be-One fields.

SBOP *See* Should-Be-One-or-Preserved fields.

SBZ *See* Should-Be-Zero fields.

SBZP *See* Should-Be-Zero-or-Preserved fields.

SCB System Control Block - an address region within the System Control Space used for key feature control and configuration associated with the exception model.

SCS System Control Space - a 4kB region of the memory map reserved for system control and configuration.

Security hole
Is a mechanism that bypasses system protection.

Set Relates to registers or register fields. Indicates the bit has a value of 1 (or bit field all 1s), or is being written with 1 or all 1s, unless explicitly stated otherwise.

SWO Serial Wire Output - an asynchronous TPIU port supporting NRZ and/or Manchester encoding.

SWV Serial Wire Viewer - the combination of an SWO and DWT/ITM data tracing capability

Self-modifying code
Is code that writes one or more instructions to memory and then executes them. This type of code cannot be relied on without the use of barrier instructions to ensure synchronization.

Should-Be-One fields (SBO)
Should be written as 1 (or all 1s for a bit field) 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 a bit field) by software if the value is being written without having been previously read, or if the register has not been initialized. Where the register was previously read, the value in the field should be preserved by writing the same value that has been previously read from the same field on the same processor.

Hardware must ignore writes to these fields.

If a value is written to the field that is neither 1 (or all 1s for a bit field), nor a value previously read for the same field on the same processor, the result is UNPREDICTABLE.

Should-Be-Zero fields (SBZ)

Should be written as 0 (or all 0s for a bit field) by software. Values other than 0 produce UNPREDICTABLE results.

Should-Be-Zero-or-Preserved fields (SBZP)

Should be written as 0 (or all 0s for a bit field) by software if the value is being written without having been previously read, or if the register has not been initialized. Where the register was previously read, the value in the field should be preserved by writing the same value that has been previously read from the same field on the same processor.

Hardware must ignore writes to these fields.

If a value is written to the field that is neither 0 (or all 0s for a bit field), nor a value previously read for the same field on the same processor, the result is UNPREDICTABLE.

Signed data types

Represent an integer in the range -2^{N-1} to $+2^{N-1}- 1$, using two's complement format.

Signed immediate and offset fields

Are encoded in two’s complement notation unless otherwise stated.

SIMD Means Single-Instruction, Multiple-Data operations.

Single-copy atomicity

Is the form of atomicity described in *Single-copy atomicity* on page A3-20.

See also Atomicity, Multi-copy atomicity.

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.

SUBARCHITECTURE DEFINED

Means that the behavior is expected to be specified by a subarchitecture definition. Typically, this will be shared by multiple implementations, but it must only be relied on by specified types of code. This minimizes the software changes required when a new subarchitecture has to be developed.

SVC Is a supervisor call.

SWI Is a former term for SVC.

Status registers

See APSR, EPSR, IPSR and xPSR.

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.

Thumb instruction

Is one or two halfwords that specify an operation for a processor in Thumb state to perform. Thumb instructions must be halfword-aligned.

TPIU Trace Port Interface Unit - part of the ARM debug architecture

UAL *See* Unified Assembler Language.

Unaligned

An unaligned access is an access where the address of the access is not aligned to the size of an element of the access.

Unaligned memory accesses

Are memory accesses that are not, or might not be, appropriately halfword-aligned, word-aligned, or doubleword-aligned.

Unallocated

Except where otherwise stated, an instruction encoding is unallocated if the architecture does not assign a specific function to the entire bit pattern of the instruction, but instead describes it as UNDEFINED, UNPREDICTABLE, or an unallocated hint instruction.

A bit in a register is unallocated if the architecture does not assign a function to that bit.

UNDEFINED

Indicates an instruction that generates an Undefined Instruction exception.

Unified Assembler Language

The assembler language introduced with Thumb-2 technology and used in this document. See *Unified Assembler Language* on page A4-4 for details.

Unified cache

Is a cache used for both processing instruction fetches and processing data loads and stores.

Unindexed addressing

Means addressing in which the base register value is used directly as the address to send to memory, without adding or subtracting an offset. In most types of load/store instruction, unindexed addressing is performed by using offset addressing with an immediate offset of 0. The `LDC`, `LDC2`, `STC`, and `STC2` instructions have an explicit unindexed addressing mode that permits the offset field in the instruction to be used to specify additional coprocessor options.

UNKNOWN

An UNKNOWN value does not contain valid data, and can vary from moment to moment, instruction to instruction, and implementation to implementation. An UNKNOWN value must not be a security hole. UNKNOWN values must not be documented or promoted as having a defined value or effect.

UNK/SBOP field

UNKNOWN on reads, Should-Be-One-or-Preserved on writes.

In any implementation, the bit must read as 1 (or all 1s for a bit field), and writes to the field must be ignored.

Software must not rely on the field reading as 1 (or all 1s), and must use an SBOP policy to write to the field.

UNK/SBZP field

UNKNOWN on reads, Should-Be-Zero-or-Preserved on writes.

In any implementation, the bit must read as 0 (or all 0s for a bit field), and writes to the field must be ignored.

Software must not rely on the field reading as zero, and must use an SBZP policy to write to the field.

UNK field

Contains an UNKNOWN value.

UNPREDICTABLE

Means the behavior cannot be relied upon. UNPREDICTABLE behavior must not represent security holes. UNPREDICTABLE behavior must not halt or hang the processor, or any parts of the system. UNPREDICTABLE behavior must not be documented or promoted as having a defined effect.

Unsigned data types

Represent a non-negative integer in the range 0 to $+2^{N}-1$, using normal binary format.

Watchpoint

Is a debug event triggered by an access to memory, specified in terms of the address of the location in memory being accessed.

Word Is a 32-bit data item. Words are normally word-aligned in ARM systems.

WO Write only register or register field. WO bits are UNKNOWN on read accesses.

Word-aligned

Means that the address is divisible by 4.

Write buffer

Is a block of high-speed memory whose purpose is to optimize stores to main memory.

WYSIWYG

What You See Is What You Get, an acronym for describing predictable behavior of the output generated. Display to printed form and software source to executable code are examples of common use.

xPSR Is the term used to describe the combination of the APSR, EPSR and IPSR into a single 32-bit Program Status Register. See *The special-purpose program status registers (xPSR)* on page B1-8.